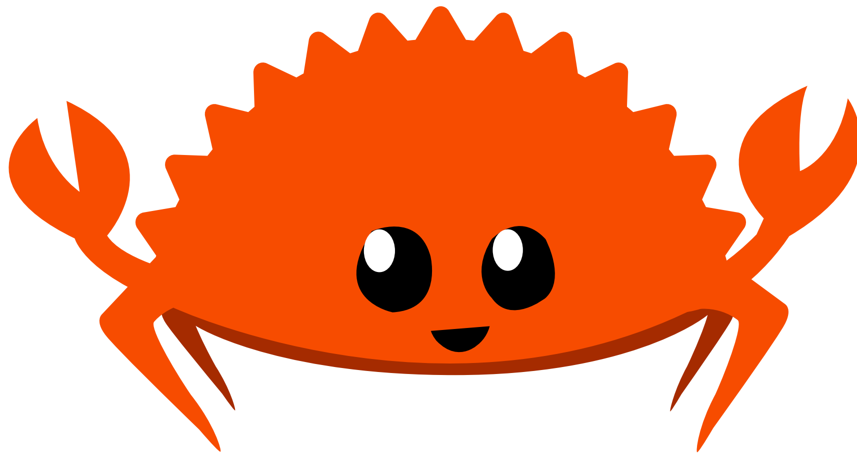
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Lock-Free Queues in Rust

Surveying, Implementing, and Benchmarking for an Improved Rust Ecosystem

Master's thesis in Computer science and engineering

Gustav Seffel

William Berg

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

Master's thesis 2025

Lock-Free Queues in Rust

Surveying, Implementing, and Benchmarking for an Improved Rust Ecosystem

Gustav Seffel
William Berg

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

Lock-Free Queues in Rust: Surveying, Implementing, and Benchmarking for an Improved Rust Ecosystem
Gustav Seffel, William Berg

© Gustav Seffel, William Berg, 2025.

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

Cover: Ferris the crab. Created by Karen Rustad Tölva. Image in the public domain (CC0).

Typeset in LaTeX
Gothenburg, Sweden 2025

Lock-Free Queues in Rust: Surveying, Implementing, and Benchmarking for an Improved Rust Ecosystem
Gustav Seffel, William Berg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Rust programming language offers strong compile-time safety guarantees and a modern approach to concurrency, positioning it as a promising candidate for developing concurrent data structures. Despite this, the ecosystem of lock-free data structures in Rust remains fragmented with significant variation in quality, performance and correctness. This thesis addresses these deficiencies by surveying the current state of lock-free concurrent queue implementations in Rust, identifying critical flaws in existing crates and benchmarking them alongside well-established C/C++ implementations. To enable consistent evaluation, we designed and developed a benchmarking framework to measure throughput, fairness, memory usage and ordering guarantees under various configurable workloads. Furthermore, we implemented several state-of-the-art lock-free queue designs in Rust and analysed their performance, highlighting challenges such as memory reclamation and limitations in Rust's type system in low-level concurrency contexts. Our results expose serious issues in popular crates and demonstrate through our implementations that high-performance, safe, and verifiable lock-free queues are achievable in Rust. In doing so, this work takes a step toward addressing the fragmentation in Rust's lock-free concurrency landscape.

Keywords: Rust, concurrent queues, lock-free, benchmarking, data structures.

Acknowledgements

We would like to express our deepest gratitude to our supervisors, Philippas Tsigas and Kåre von Geijer, for their invaluable support and guidance throughout this project. Their insights and feedback have been instrumental in shaping the direction and quality of our work. In particular, we are profoundly thankful to Kåre von Geijer for his continuous and hands-on involvement during all stages of the project. His readiness to assist with both technical consultations and broader conceptual discussions has been a cornerstone of our progress, and we are sincerely appreciative of the time and expertise he has generously shared with us.

We would also like to express our gratitude towards our examiner, Yehia Abd Alrahman, who guided us through valuable feedback during the work of this thesis.

Gustav Seffel, William Berg, Gothenburg, 2025-06-17

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Purpose 2
1.2 Limitations 3

2 Preliminaries **5**
2.1 FIFO and LIFO queues 5
2.2 The Rust programming language 7
2.2.1 Unsafe Rust 7
2.2.2 Foreign Function Interface 8
2.3 Concurrent data structures 9
2.3.1 Non-blocking data structures 9
2.3.2 Compare-and-swap and fetch-and-add 10
2.3.3 Lock-free memory reclamation 10
2.3.4 Linearizability 11
2.4 Benchmarks in software engineering 11

3 Related work **13**
3.1 Lock-free concurrent queues 13
3.2 Benchmarking 15
3.3 Rust for concurrent algorithms 16

4 Methods **19**
4.1 Benchmark criteria and test design 19
4.1.1 Irregular parallelism 19
4.1.2 Throughput 20
4.1.3 Fairness 20
4.1.4 Memory usage 20
4.1.5 Ordering 21
4.2 Implemented benchmarks 22
4.2.1 Producer-Consumer 22
4.2.2 Enqueue-Dequeue 23
4.2.3 BFS 24

4.3 Implementation of Benchmarking Framework . . . 24
4.3.1 Relevance . . . 24
4.3.2 Reproducibility . . . 25
4.3.3 Fairness in benchmarking . . . 26
4.3.4 Verifiability . . . 26
4.3.5 Usability . . . 26
4.4 Integration of queues into framework . . . 27
4.4.1 Creation of Rust queues . . . 29
4.5 Specifications of test bed . . . 32
4.6 Licence . . . 32

5 Results **33**
5.1 Performance, fairness and memory . . . 33
5.1.1 lockfree::Queue crashing . . . 40
5.1.2 Queue optimisations . . . 41
5.2 BFS benchmarks . . . 42

6 Discussion **43**
6.1 Results summary . . . 43
6.2 Benchmarking framework . . . 43
6.2.1 The critical flaw of lockfree::queue . . . 45
6.2.2 SCC/SCC2 poor performance in freeing memory . . . 48
6.2.3 Wfqueue: Hidden order violations . . . 50
6.2.4 Bbq blocks threads . . . 50
6.3 Rust's queue performance potential . . . 52
6.4 State of the Ecosystem . . . 52
6.4.1 Common Rust concurrency pitfalls . . . 53

7 Conclusion **57**
7.1 Future work . . . 58

Bibliography **59**

A File structure **I**

B Queue implementation example **III**

List of Figures

2.1 Visualisation of a FIFO queue. 5
2.2 Visualisation of a LIFO queue. 6

5.1 Throughput (Ops/s) and fairness comparison of bounded queues with $\tau = 0.3$, 0.5, and 0.7. For each τ value: Left: Throughput across multiple thread counts. The y-axis uses a logarithmic scale. Right: Fairness across multiple thread counts. 34
5.2 Throughput across multiple thread counts for Wfqueue against the C++ queues LPRQ, LCRQ and FAAAQueue ($\tau = 0.5$). The y-axis uses a logarithmic scale. 35
5.3 Throughput (Ops/s) comparison of unbounded queues with $\tau = 0.3$, 0.5, and 0.7. The y-axis uses a logarithmic scale. For each τ value: Left: Existing ecosystem and C++ implementations. Right: Our Rust implementations (highlighted) shown in context with the same queues (greyed) for reference. 36
5.4 Fairness across multiple thread counts for all implemented unbounded queues with $\tau = 0.3$, 0.5, and 0.7. For each τ value: Left: Existing ecosystem and C++ implementations. Right: Our Rust implementations (highlighted) shown in context with the same queues (greyed) for reference. 38
5.5 Mean peak memory usage of the bounded queues at 36 threads in the Enqueue-Dequeue benchmark ($\tau = 0.5$). The y-axis uses a logarithmic scale. 39
5.6 Mean peak memory usage of the unbounded queues at 36 threads in the Enqueue-Dequeue benchmark ($\tau = 0.5$). The y-axis uses a logarithmic scale. 39
5.7 A plot of several benchmarks on the lockfree::Queue. The y-axis uses a logarithmic scale. 40
5.8 Throughput (Ops/s) of an unoptimised and an optimised version of LPRQ written in Rust and the throughput of a C++ LPRQ implementation ($\tau = 0.5$). The y-axis uses a logarithmic scale. 41
5.9 Throughput (Ops/s) of an unoptimised and an optimised version of LCRQ written in Rust and the throughput of a C++ LCRQ implementation ($\tau = 0.5$). The y-axis uses a logarithmic scale. 41
5.10 Comparing unbounded and bounded queues mean time to complete BFS. Lower is better. Graphs available at NetworkRepository [64]. . . 42

6.1 A flamegraph where the horizontal axis represents execution time and the vertical axis indicates call stack depth. Colours are used solely for visual distinction between stack frames and carry no meaning. The graph illustrates the significant time disparity between benchmark execution and memory deallocation operations. 49

List of Tables

4.1 The configurable options for the Producer-Consumer benchmark. . . . 23
4.2 The configurable options for the Enqueue-Dequeue benchmark. 23
4.3 The configurable options for the BFS benchmark. 24
4.4 Queues included in our benchmarking framework from crates.io. . . . 28
4.5 C/C++ queues included in the benchmarking framework. 29
4.6 Our own queue implementations included in the benchmarking framework. 29

6.1 Major bugs found through the use of the benchmarking framework. . 44
6.2 Findings made through the use of the benchmarking framework. . . . 45

1
Introduction

The programming language Rust is a somewhat new general-purpose language that puts high emphasis on performance, type safety and concurrency, targeting the same low-level domain as the popular system languages C and C++, but with modern safety guarantees [1]. Instead of having developers manage memory manually, Rust uses a system called *ownership*. In the ownership system, the Rust compiler tracks exactly which part of your code owns each piece of data. When you want to use data you do not own, you have to "borrow" it. Rust's "borrow checker" keeps track of this and ensures all borrowed references will be valid when the program runs. By doing this, Rust can catch many difficult-to-manage errors at compile-time instead of at run-time. This makes Rust a general-purpose language that attempts to achieve the performance C and C++ are known for, but with more safety and support from the compiler.

Concurrency refers to a system's ability to manage multiple tasks that progress during overlapping time intervals [2]. This does not necessarily mean that tasks are executed at the exact same time (that would require parallelism) but rather that they can be interleaved, with the system switching between them as needed. Compared to sequential execution, where tasks run one after the other, concurrent execution typically enables more responsive and efficient programs, particularly in systems that handle multiple independent operations.

Rust provides what is known as "Fearless concurrency", which via the borrow-checker and Rust's type system, is designed to make writing concurrent code a safe and predictable process instead of an unpredictable one [1], [3]. By enforcing strict rules around ownership and access, Rust eliminates some concurrency-related bugs at compile-time, such as data races. This way, incorrect code will refuse to compile, and the compiler will present error messages explaining the problem. As a result, one can fix the code while working on it, rather than trying to reproduce elusive run-time errors later.

Despite these language-level innovations, the ecosystem of concurrent data structures in Rust remains surprisingly fragmented, with implementations varying widely in quality, correctness, and performance characteristics. Rust's build system, Cargo, consists of crates, which can be seen as libraries, where any developer can publish code to the Rust ecosystem. This, however, creates fragmentation. As very few crates include documentation on the underlying data structure used, or empirical evaluation of it, questions about the quality, reliability, and consistency of the different crate

implementations are raised. Take the crate "lockfree" [4] as an example. As we will show in this paper, the `lockfree::Queue`, a supposedly lock-free queue in the lockfree crate, is in reality incorrectly implemented and is contrary to its name, not in fact lock-free. It is not even a properly implemented concurrent queue.

Abdi et al. [5] in 2024 investigated the capabilities of writing concurrent algorithms in Rust and benchmarked several concurrent data structures. However, these efforts lie mainly in benchmarking their implementations, not in surveying the Rust ecosystem. Furthermore, they benchmark through three main criteria to explore the fearlessness of Rust. These are: Data Structure, Operator and Set of tasks. This set of categories covers a large set of parallelism, however, they did not explore the lock-free environment in Rust, thus leaving an important area of concurrency largely unexamined.

Lock-free programming is a large and well-established field, as evidenced by the extensive research literature [6]–[9] and its use in commercial applications where scalability and high throughput are needed, such as task schedulers [10] and real-time systems [11]. Among lock-free data structures, queues stand out as one of the more fundamental and widely studied due to their central role in many concurrent systems, such as operating systems and databases [12]. The challenge of implementing lock-free queues lies not only in ensuring correctness under concurrent access without traditional synchronisation methods but also in balancing trade-offs between throughput, scalability and memory reclamation. As a result, there have been a variety of designs presented in literature, such as the Michael-Scott Queue [13], the LCRQ [14] and the newly added LPRQ [15], each with its own features and patterns that make them unique.

Despite lock-free data structures being an extensive area, the Rust ecosystem completely lacks any structured and comprehensive evaluation of such queues. The few implementations that exist are scattered across crates of varying quality, and their performance and correctness are often undocumented or unverified. This makes it difficult for developers to choose the right implementation of a data structure for their specific needs.

To address this existing gap, we aim to develop a benchmarking suite for concurrent queues in Rust to be able to survey, implement, and benchmark for an improved ecosystem.

1.1 Purpose

Given the challenges posed by the fragmented nature of Rust's crate ecosystem, this thesis aims to improve the landscape of concurrent data structures in Rust. In addition, it will attempt to address the difficulties developers face in selecting appropriate and reliable concurrent queues.

The core challenge in this thesis lies in developing a comprehensive benchmarking framework that can objectively compare different concurrent queue implementations. This requires creating a proper way of evaluating data structures across multiple

dimensions, enabling developers to make informed decisions based on concrete performance metrics, reliability assessments, and scalability characteristics.

In our thesis, we aim to address these issues by answering the following scientific questions:

- How does one develop an effective benchmarking framework in Rust that enables consistent and meaningful comparisons between different concurrent data structures?
- What kind of tests and evaluation criteria are essential in capturing the trade-offs between different queue designs, and how can these be used to assess the overall maturity of Rust's concurrent ecosystem?
- To what extent can the benchmarking framework identify gaps in Rust's concurrent ecosystem, and can we help resolve the gaps by contributing state-of-the-art implementations?

By addressing these questions, we aim to not only provide a benchmarking framework for immediate use but also to shed some light on the current state of Rust's concurrent ecosystem, highlighting both its strengths and weaknesses.

1.2 Limitations

If a non-concurrent data structure is incorrectly implemented, it often fails to compile due to type or syntax errors, or any issues that do arise are typically easier to detect during runtime. However, this is not the case for concurrent data structures, where subtle and hidden faults can still exist even if the code compiles, such as race conditions, livelocks, deadlocks, and so on [5], [16]. Given the complexity and critical importance of properly testing the correctness of concurrent data structures, our benchmarking framework will prioritise their evaluation.

Given their foundational importance [17] and relative simplicity, the focus will be on concurrent queues, categorised into (unbounded and bounded) First In First Out queues (FIFO) and Last In First Out (LIFO) queues (stacks).

2

Preliminaries

This chapter provides the necessary preliminaries to understand the work presented in this thesis. It covers core concepts related to queues, the Rust programming language, benchmarking methodology, and concurrency paradigms.

2.1 FIFO and LIFO queues

Queues are a data structure commonly used for tasks such as processing requests in a web server [18] or scheduling operations in an operating system [10], [19]. How a queue operates differs depending on whether it is a First-In First-Out (FIFO) or a Last-In First-Out (LIFO) queue.

In a FIFO queue, elements may only be inserted at the back of the queue, and elements may only be removed from the front. These operations are called enqueue and dequeue. Due to this specification, FIFO queues release their elements in the order they arrived [20]. A visualisation of a FIFO queue can be seen in Figure 2.1.

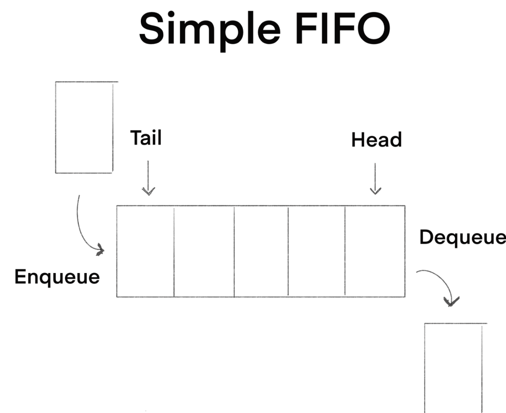


Figure 2.1: Visualisation of a FIFO queue.

In a LIFO queue, elements may only be inserted and removed from one end of the queue. An insertion in a LIFO queue is called a push, and a removal is called a pop. Due to this specification, LIFO queues release their elements in the reverse order of their arrival. LIFO queues are often referred to as **stacks** [20]. A visualisation of a LIFO queue can be seen in Figure 2.2.

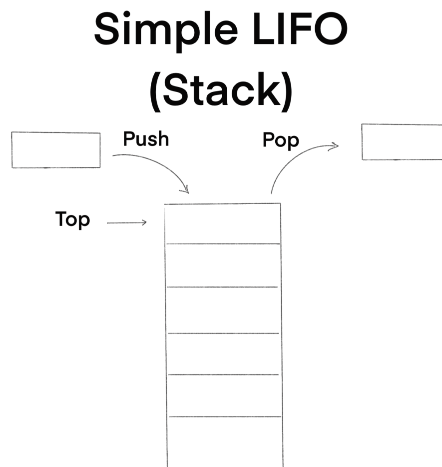


Figure 2.2: Visualisation of a LIFO queue.

Another important implementation consideration is the size constraint of a queue. A queue's size can be either *bounded* or *unbounded*.

A bounded queue has a fixed maximum size, that is, it can only hold a limited number of elements, typically implemented via a fixed-size type like an array. Once the queue is filled, no more elements can be added until more space is freed [21]. This design prevents excessive memory usage by restricting the queue's growth, ensuring that it does not consume more resources than allocated. However, it also introduces certain limitations, such as the potential for blocking new elements when the queue is full and the need for overflow handling to manage scenarios where incoming data exceeds the queue capacity.

An unbounded queue does not have a fixed size and can dynamically grow as long as there is memory. It expands as needed when new elements are added [21], these are typically implemented via dynamic types such as a linked list. One key advantage of this design is that it eliminates the risk of overflows, allowing for continuous data insertion without predefined limits. However, this flexibility comes with potential drawbacks, as the queue can consume excessive memory and suffer performance issues if it grows uncontrollably.

2.2 The Rust programming language

Rust is a general-purpose programming language emphasising performance, memory safety and type safety. It is designed to inherit C's performance characteristics and feature set while eliminating safety vulnerabilities through comprehensive and rigorous compile-time checks [22]. Rust has no garbage collector; it instead uses a principle called *ownership* to decide when to free memory. Each value has one *owner* and is freed when its owner's lifetime ends, usually when it goes out of scope. Ownership can be borrowed or transferred, and multiple immutable references to the same data are allowed simultaneously. However, you can not have a mutable reference and an immutable reference to the same value, or several mutable references to the same value. Rust checks these rules while compiling and can through this achieve performance on par with unsafe languages like C, but with much stronger safety guarantees.

Rust also provides "fearless concurrency" [1], meaning that writing concurrent code should be a safe and predictable process rather than a source of fear via difficult-to-debug errors. This is achieved through Rust's ownership model and type system, which enforces memory safety and prevents concurrency issues (mainly data races) at compile-time. By leveraging these features, many potential errors that would typically manifest as unpredictable runtime bugs in other languages are instead caught as compile-time errors.

2.2.1 Unsafe Rust

While these stringent compile-time checks mitigate programming risks, they simultaneously impose significant constraints on what kind of code you can write. Sometimes the compiler can not determine completely if the code is safe and thus rejects it [1]. This leads to developers sometimes being compelled to use *unsafe* Rust, a language subset that provides low-level memory manipulation capabilities similar to C. In unsafe Rust, you can do five things that are not possible in "safe" Rust:

- Dereference a raw pointer.
- Call an unsafe function or method.
- Access or modify a mutable static variable.
- Implement an unsafe trait.
- Access fields of a union.

To use unsafe Rust, create an `unsafe{}` code block. In that code block, you can utilise the features of unsafe Rust. An example of unsafe Rust code can be seen in Listing 1.

```
fn deref_ptr() {
    let mut num = 5;

    let r1 = &raw const num;
    let r2 = &raw mut num;

    unsafe {
        // Dereferencing a raw pointer is unsafe
        // as it can cause undefined behaviour.
        println!("r1 is: {}", *r1);
        println!("r2 is: {}", *r2);
    }
}
```

Listing 1: Example of unsafe Rust code.

It is important to note that the unsafe keyword does not bypass all of Rust's safety mechanisms. Instead, it specifically permits the use of the five previously mentioned features without compiler-enforced memory safety checks. The compiler still enforces Rust's other safety guarantees, but the responsibility for memory safety within unsafe blocks shifts entirely to the developer [1].

Another important aspect of unsafe Rust is that you can have unsafe code within a safe interface. Meaning, you can have unsafe code inside a safe function. This approach allows developers to encapsulate potentially dangerous operations behind a safe abstraction layer, hiding the unsafe implementation details from users. When designed properly, these safe interfaces guarantee that all preconditions for unsafe operations are checked and enforced, ensuring memory safety despite using unsafe features internally. This pattern is common and is used in Rust's standard library, where many safe functions and methods contain unsafe code blocks [1].

One real-world scenario that requires unsafe code to function is inter-operability between other programming languages, such as C or C++ to Rust. This is done through Rust's Foreign Function Interface.

2.2.2 Foreign Function Interface

Foreign Function Interface (FFI) is a mechanism that allows code written in one programming language to call and interact with code written in another programming language [1]. In this work, this is used to run C/C++ code in Rust. However, extra precautions are needed since Rust provides fearless concurrency while C/C++ often expose interfaces that are not thread-safe. Any function that takes a pointer as an argument may be invalid in Rust's safe memory model, as the pointer could be dangling. Therefore, all FFI calls must be wrapped in an `unsafe{}` block, serving as a promise to the compiler that the code is safe [3]. See Section 2.2.1 for more on unsafe Rust.

2.3 Concurrent data structures

Concurrency refers to the property of a system in which multiple processes are executed within overlapping time intervals [2]. However, concurrency does not necessarily mean that tasks are literally running simultaneously (which requires parallelism and multiple cores), but rather that tasks are in progress simultaneously, often by switching between tasks. Unlike sequential execution, where tasks are performed one after another, concurrent execution enables tasks to progress independently, improving performance.

Traditionally, concurrent data structures are achieved using blocking synchronisation techniques such as locks or semaphores, which enforce mutual exclusion. While effective, these approaches can introduce problems like deadlocks, priority inversion, and performance bottlenecks under high contention.

To address these issues, the concept of non-blocking data structures emerged. These data structures avoid locks entirely by using atomic operations to manage shared memory. Non-blocking data structures are categorised by their progress guarantees as lock-free and wait-free.

In this work, we use the terms concurrency and parallelism interchangeably, focusing on systems where multiple threads operate simultaneously. While distinctions exist, with concurrency emphasising switching between threads of execution and parallelism emphasising true simultaneous execution, the concepts overlap in our context of multithreaded performance benchmarking.

2.3.1 Non-blocking data structures

Non-blocking strategies allow multiple threads to access shared data structures without waiting for locks. This approach is particularly advantageous in high-performance and real-time applications because it minimises contention, removes the risk of deadlocks, livelocks, and improves overall responsiveness. Non-blocking algorithms include both lock-free and wait-free approaches.

A data structure is lock-free if and only if it guarantees that at least one thread will make progress in a finite number of steps, regardless of how other threads behave. This ensures system-wide progress guarantee, prevents deadlocks and livelocks, but not necessarily individual thread starvation [23]. Lock-free implementations rely on atomic operations, such as compare-and-swap (CAS) or fetch-and-add (FAA) (see Section 2.3.2) to coordinate access to shared resources in ways not requiring threads to wait for one another.

A stronger progress guarantee is provided by wait-freedom, which ensures that all threads complete their operation within a finite number of steps, regardless of other threads' behaviour. This eliminates the possibility of individual thread starvation [23]. Most wait-free algorithms achieve this by employing the "helper" method. Meaning that threads whose concurrent operations would prevent the operation from succeeding are turned into helpers, who help the operation to complete, thus ensuring progress for all threads [6], [24].

2.3.2 Compare-and-swap and fetch-and-add

In the realm of lock-free concurrency, the need to update shared variables is essential, but non-trivial. One of the most fundamental tools for this is the compare-and-swap (CAS) operation. The CAS operation (at its simplest form) takes three arguments: the address of a shared data item, an old value of the shared data item, and a new value. The operation checks if a value at the address is equal to the old expected value, and if it is, it atomically swaps the value to the new value [25], [26], otherwise it does nothing. In addition to CAS, there also exists CAS2, which updates two contiguous memory locations [15]. CAS2 is non-portable and is only available on the x86 architecture.

However, the CAS operation suffers from limitations in practical implementations. CAS suffers from consistent failure under high contention due to the constant changing of values.

Another atomic operation used in lock-free concurrency is fetch-and-add (FAA). FAA atomically increments a value at address x by an amount a, and returns the original value before the increment. FAA never fails to complete its operation, as opposed to CAS. Current research favours the fetch-and-add operation, which demonstrates superior scalability and performance under load [15].

2.3.3 Lock-free memory reclamation

In lock-free data structures, memory management is inherently very challenging [27]. While threads may work concurrently on shared data, it is often unclear when to reclaim memory. For example, when a thread dequeues a node from the queue, other threads could potentially still hold references to the node. Due to this, the node can not be freed. It can only be freed once no other thread holds a reference to it. If a thread deallocates memory that another thread is accessing, it can lead to errors and undefined behaviours.

Hazard pointers are a method for memory reclamation in dynamic, lock-free data structures that guarantees progress and ensures safe memory access [8], [27]. The core idea is to allow threads to signal which memory locations they are currently accessing, thereby preventing premature reclamation of those objects. Each reader thread maintains a hazard pointer, a shared, single-writer/multi-reader pointer that it uses to indicate interest in a specific memory location publicly. Before a thread accesses a shared object, it assigns the address of that object to its hazard pointer. This effectively announces to all other threads that the thread is currently reading this object, and while replacement is permissible if necessary, modification or deallocation of the object during this period of access must be avoided.

Another method for memory reclamation is epoch-based memory reclamation. This reclamation scheme works by keeping a global epoch counter to keep track of the progress of operations across all threads in the system [23]. When a thread begins a shared memory operation, it registers the current epoch, indicating that it might access any objects created or deleted until that point. When an object is deleted, it is not immediately freed. Instead, it is added to a limbo list associated with the

current epoch. This limbo list acts as a holding area, preventing the object from being deallocated while other threads might still reference it. Periodically, the system checks whether all threads have moved on to newer epochs, meaning no thread is still working in the epoch in which the object was retired, or the one before it. Once this condition is met, reclaiming the memory from every object in the limbo list associated with that epoch and incrementing the epoch counter is safe. However, if a thread stalls during a shared-memory operation, then memory in the limbo list will never be reclaimed. Other threads will then only be able to continue running until the memory limit is reached.

2.3.4 Linearizability

Linearizability is a correctness condition for concurrent objects, requiring that each method call should appear to take effect instantaneously at some point between when it begins and when it completes [28], [29]. A concurrent execution satisfies linearizability if its outcome is equivalent to a legal sequential computation. More informally, if one method call proceeds another, then the earlier call must take effect before the later call.

2.4 Benchmarks in software engineering

Kistowski et al. define a benchmark as:

"In the realm of software engineering, a benchmark is a standardised test to evaluate and compare the performance of different software or algorithms." [30]

Beyond performance comparisons, a benchmark is often used to further evolve certain research agendas by providing a standardised way of comparing [31]. There are several benefits to benchmarking, especially regarding scientific maturity. Creation and widespread adoption of a benchmark is often followed by rapid technological advancement and stronger community collaboration. However, for a benchmark to be relevant, it is also essential that it represents a task that is expected to be solved in actual practice (this will be further discussed in Section 4.1).

3

Related work

This chapter provides an overview of previous research, tools and methodologies that are relevant to this project. This chapter mainly has two key purposes. Firstly, to contextualise current work done in relevant or similar areas, where we highlight the landscape in which our work will be done, and secondly, to identify gaps. By identifying previous work, we can establish areas that need to be highlighted and how our project differs from other existing projects.

3.1 Lock-free concurrent queues

Concurrent FIFO queues are widely used in parallel applications and operating systems and have been a topic of active research for decades. The lock-free queue by Michael and Scott from their paper in 1996 [13], called the Michael-Scott Queue (MSQueue), is considered a classic lock-free queue. It is built upon a linked list and has two pointers to the head and tail of the list. Each node in the queue has an atomic pointer to the next node in the queue. The dequeue operation atomically removes elements by advancing the head pointer to the next node via CAS within a retry loop until successful. When enqueueing a new node, the current tail's next pointer is first updated, and then the tail of the queue is updated, both done using CAS in a retry loop. The queue does not scale well due to performance degradation under high contention, caused by the CAS operation, which will often fail under heavy contention [6]. For discussion of work that predates theirs, see their extensive survey in [13]. Several works [32]–[34] try to improve upon the MSQueue to make it scale better; however, they still have the same problem with a highly contented CAS as the MSQueue [14].

The MSQueue is an *unbounded* queue, meaning it can grow to accommodate any number of items. In contrast, cyclic array queues are *bounded*, containing a fixed maximum number of elements. These bounded queues can be implemented using pre-allocated arrays with positioning counters that determine where items are stored and removed. These counters can potentially be updated by the atomic operation fetch-and-add (FAA), which does not fail like CAS. This approach can significantly improve algorithm scalability, though implementing these cyclic array queues remains non-trivial. A key difficulty is accurately determining whether the queue is full. The queues presented by Gottlieb et al. [35] and Freudenthal and Gottlieb [36] can reach inconsistent states due to their use of FAA when tracking queue size, resulting in the

queues not being linearizable. Blelloch et al. [37] achieved a linearizable queue using FAA on positioning counters, despite temporary inconsistent states during counter updates, though their implementation did not allow concurrent enqueue and dequeue operations. Later implementations by Tsigas and Zhang [17], Colvin and Groves [38], and Shafiei [39] avoided inconsistent states while maintaining linearizability. These queues, however, use CAS and therefore suffer the effects of the CAS operation at high contention. Wang et al. [12] presents Block-based Bounded Queue (BBQ), which splits the underlying array into several blocks. The main focus of BBQ is to reduce the interference that enqueue and dequeue operations have on each other. Their evaluations show that their queue outperforms several industrial cyclic array queues.

In 2013, Morrison and Afek in [14] present Linked Concurrent Ring Queue (LCRQ), a non-blocking linearizable FIFO queue. It combines list-based queues, such as the MSQueue, with cyclic array (concurrent ring) queues. LCRQ is essentially an MSQueue where a node is a concurrent ring queue (CRQ). When a CRQ is filled up, it becomes *closed* to further enqueues, and instead appends a new CRQ to the linked list. Compared to prior concurrent ring queues [17], [38], [39], an operation on the CRQ in the common case only accesses the CRQ's head or tail, but not both. This approach halves the synchronisation overhead by eliminating the need to simultaneously contend with both the head and tail, which are typically the performance bottleneck. As well as that, most of the time, contention is on an FAA operation instead of a CAS operation, as the CRQs use FAA to increase their positioning counters. This means LCRQ is much less susceptible to the performance degradation that high contention on CAS causes. LCRQ is one of the fastest concurrent queues to date. In their experiments running on a single processor, from ten threads and onwards, it outperforms MSQueue by more than three times. However, LCRQ relies on CAS2, an operation not available on all architectures, and programming languages such as Java, Kotlin and Go [15]. This makes the LCRQ algorithm much less portable than algorithms using only CAS.

Yang and Mellor-Crummey presented in 2016 the first wait-free linearizable and fast FIFO queue that is based on FAA [6]. Their design uses an infinite array approach with head and tail indices, employing a fast-path-slow-path methodology to guarantee wait-freedom. Proposed by Kogan and Petrank in 2012, fast-path-slow-path is a methodology to create practical wait-free objects [24]. Operations first attempt the fast-path using FAA operations, falling back to a guaranteed-to-complete slow-path after repeated failures, thus ensuring both performance and progress guarantees without relying on CAS2. Yang and Mellor-Crummey's queue demonstrated comparable performance to LCRQ, outperforming it in some scenarios while underperforming in others.

In 2016, Ramalhete introduced FAAArrayQueue, a portable linearizable unbounded lock-free queue [40]. Similar to LCRQ, it consists of a linked list where each node is a bounded queue; however, in the FAAArrayQueue, they are non-cyclic. Unlike LCRQ, FAAArrayQueue achieves portability by relying exclusively on FAA and CAS operations. Performance-wise, FAAArrayQueue demonstrates comparable efficiency to LCRQ, occasionally outperforming it in certain benchmarks. However, FAAAr-

rayQueue's inability to reuse entries in its underlying bounded queues, a capability that LCRQ has, prevents it from consistently surpassing LCRQ's performance.

In 2023, Romanov and Koval present the Linked Portable Ring Queue (LPRQ) [15], a portable modification of the LCRQ algorithm which does not utilise CAS2, instead only relies on the atomic operations CAS and FAA. Their experiments show that LPRQ provides the same performance as LCRQ.

3.2 Benchmarking

Various researchers have explored the essential properties that effective benchmarks should have. While there is overlap in their findings, each perspective offers insights into benchmark design and evaluation.

Sim et al. [31] identify seven key properties that successful benchmarks should have: accessibility, affordability, clarity, relevance, solvability, portability, and scalability. "Accessibility" ensures the benchmark is openly available to anyone who wishes to use it, as well as easy to use. "Affordability" means the cost of using the benchmark should be proportionate to the benefits. "Clarity" requires the benchmark to be easy to understand and self-contained. "Relevance" ensures the tasks in the benchmark are ones the system being benchmarked is expected to handle. "Solvability" means tasks should be achievable. "Portability" ensures the benchmark functions across different platforms and environments. Finally, "scalability" allows the benchmark to accommodate varying system capabilities and resources.

A somewhat different perspective is offered by Huppler [41], who propose five properties while acknowledging the inherent trade-offs in benchmark design. They argue that no benchmark can perfectly satisfy all criteria simultaneously, however that most good benchmarks excel in one or two of the properties, and accommodate the other three. Their properties are that the benchmark should be relevant, repeatable, fair, verifiable, and economical. For Huppler, "relevant" refers to the fact that the benchmark should measure characteristics that matter to real-world applications. "Repeatable" ensures results are consistent when running the same test configuration several times. "Fair" means the benchmark should not favour certain approaches or implementations. "Verifiable" ensures results can be independently verified by third parties. "Economical" means that it should be affordable to run the benchmark.

Kistowski et al. [30] conducted a comparative analysis of existing studies of benchmarking, including the two previously mentioned studies. They identify five essential properties, largely aligning with Huppler but replacing "economical" with "usability". "Usability" emphasises that the benchmark should be easy to run in the user's test environment. Kistowski et al. state that the most important property is that of relevance. Even if perfect in every other way, if the output of the benchmark is irrelevant, then it does not matter.

3.3 Rust for concurrent algorithms

There was an article by Abdi et al. [5] in SPAA in 2024 that explored Rust's capabilities for writing concurrent algorithms. In it, the authors ported the *problem-based benchmark suite*[1] [42], [43] from C++ to Rust, offering an analysis of Rust's concurrent programming capabilities. The article showed that there were notable challenges with porting the benchmark to Rust, as well as problems with irregular memory access patterns, which often lead to reduced performance compared to its C++ counterpart. However, they also noted that Rust made it easier to detect bugs in the code, even finding a bug in the original code. In addition, unsafe code blocks could be encapsulated inside small pieces of code, allowing developers to control and minimise the scope of low-level memory operations more precisely while maintaining the overall safety guarantees of the Rust language.

In [44], Saligrama et al. investigate Rust's capabilities when developing concurrent data structures. The authors implemented several different concurrent hashmap designs, with increasing levels of sophistication and then evaluated their performance. They identify several good aspects and several bad aspects of developing concurrent code in Rust, as well as some aspects that make the code less readable and/or understandable. Most notably among the good aspects they find that Rust's memory management is very useful when writing safe code and that the ownership and mutability rules once learned result in you producing better code. They also note that since you have to explicitly mark unsafe code with an *unsafe* block, it forces the developer to take note that they are doing something potentially dangerous, and it makes it easier to find concurrency bugs later on in development. A key drawback, however, is that Rust's memory management can become challenging to navigate when dealing with unsafe code.

In 2020, Qin et al. [22] conducted an empirical study on how unsafe code is used, modified, and encapsulated in Rust. They examined memory safety issues and investigated concurrency bugs, including both blocking and non-blocking ones. They studied, in total, 850 unsafe code usages, 70 memory-safety issues, and 100 thread-safety issues. The investigation was structured around these central questions: Firstly, how unsafe code is used, changed, and encapsulated. Secondly, memory-safety issues in real Rust programs by inspecting bugs. And finally, concurrency bugs, including non-blocking and blocking bugs. Based on their findings, the authors provide answers to the following questions:

1. When and why to use unsafe code?
2. How to properly encapsulate unsafe operations?
3. How to change unsafe code to safe code?

The authors note that while it is considered good practice to minimise the use of unsafe code, it is sometimes necessary, for instance, when interfacing with low-level hardware or OS components, or for performance-critical operations. When unsafe

[1] The problem-based benchmark suite is a collection of benchmarks designed to test problems on different platforms and using different algorithms.

code is required, they emphasise the importance of careful encapsulation to contain potential risks. Furthermore, they outline strategies for eliminating unsafe code when possible, such as replacing unsafe reads and writes to shared variables with atomic operations.

The Rust ecosystem has a notable deficiency in comprehensive benchmarking frameworks for concurrent queues. Beyond the previously discussed ported benchmarking suite, the landscape consists primarily of a couple general-purpose benchmarking libraries [45]–[47]. Several other general-purpose benchmarking libraries can be found on crates.io, which is the Rust community's crate registry [48], such as [49] and [50], but all implementations we found suffer from low popularity, inadequate documentation and low to no ongoing maintenance. Our investigation revealed no widely embraced benchmarking solutions that meet standards for active development or feature completeness specifically for concurrent queue implementations. A benchmarking library specifically for queues is available, though it lacks popularity and active management, with its latest update occurring in 2019 [51]. This neglected segment of the Rust ecosystem presents substantial opportunities for improvement and innovation.

4

Methods

This chapter explains the methods used for developing and evaluating the benchmarking framework and the integration and implementation of queues. Initially, research about benchmarking, benchmarks, concurrent queues and Rust was done to understand the problem. Building on this research, the benchmarking framework was developed. After this, an evaluation of the benchmarking framework was done, based on several criteria found in the literature. Following this, several open-source queue implementations found on crates.io and a few C++ queues were tested using our benchmarking framework to evaluate the framework, as well as to evaluate the Rust concurrent queue ecosystem. Then we implemented known lock-free concurrent queues from literature ourselves to evaluate the usefulness of the benchmarking framework, and also to evaluate the process of implementing concurrent algorithms in Rust.

4.1 Benchmark criteria and test design

The choice of criteria and tests was made through examining previous work (including but not exclusive to) [5], [6], [14], [15], [40], [52]. In the following subsections, we describe each benchmark criterion used in our framework, the motivation behind each one and the way we apply these evaluations.

4.1.1 Irregular parallelism

Irregular parallelism is the definition where tasks and workloads follow an unpredictable, non-uniform pattern. Unlike regular parallelism, where work can be split into equally sized, independent tasks (such as processing evenly divided arrays) [53]. As Abdi et al. mentioned in [5], there is a lack of benchmarks that test irregular parallelism. Therefore, we incorporated the option to also test irregular parallelism in our benchmark framework. For example, in the form of random consumer/producer values (to simulate different work environments), as well as the ability to choose how many of the threads will produce/consume in the model (to simulate high/low workloads).

4.1.2 Throughput

Throughput is a performance metric that quantifies how efficiently a data structure handles operations over time and is defined as the number of operations a data structure performs per second. A high throughput indicates that the data structure can process a large number of operations efficiently [54]. Contrary, a low throughput means that the data structure can create bottlenecks, resulting in delayed responses and increased latency.

Throughput is particularly important when evaluating data structures in concurrent programming. It provides an insight into how efficiently it utilises its computing resources. Benchmarks often measure throughput under a lot of different conditions such as, varying levels of concurrency, different workloads and different sizes to provide comprehensive information about the data structure.

4.1.3 Fairness

Fair resource allocation is a critical consideration in modern multithreaded systems. Without fairness, some threads may suffer from starvation, thus being indefinitely delayed, while others dominate access to the shared resources [55]. This behaviour can lead to a multitude of problems such as unexpected behaviours, unpredictable performance or even system failures.

The definition of fairness is a bit vague in computer science and usually changes depending on the area being focused on [56]. However, a relevant definition of fairness in our context for concurrent data structures was mentioned in the paper by Cederman et Al. [55]. Here, fairness is defined by comparing the minimum number of operations performed by any thread against the average number of operations across all threads. This comparison helps identify cases of starvation or less served threads. To detect the opposite cases, where certain threads are disproportionately favoured, we can compare the average to the maximum number of operations among all threads. Since our goal is to measure any unfair behaviour, we use as a fairness measure, the minimum of the values mentioned above. More formally:

$$fairness_{\Delta t} = \min\left\{\frac{N \cdot \min(n_{i_{\Delta t}})}{\sum_i n_{i_{\Delta t}}}, \frac{\sum_i n_{\mathrm{i}_{\Delta t}}}{N \cdot \max(n_{i_{\Delta t}})}\right\}$$

Where $n_{i_{\Delta t}}$ is the number of successful operations by the thread i during the time interval Δt. This equation will give a number between zero and one, where one is the most fair, meaning that all threads do the same amount of operations, and a value of zero indicates that at least one thread is completely starved.

4.1.4 Memory usage

Another important aspect to evaluate is the memory usage. Memory tracking allows us to analyse how much memory a data structure allocates and deallocates during certain configurations and workloads, thus providing valuable information about the data structure's memory efficiency.

To accurately track memory usage, we utilise a pre-made memory allocator named jemalloc [57], a high-performance memory allocator designed for multi-threaded applications. While jemalloc is a general-purpose memory allocator, it provides efficient and predictable memory allocation patterns, making it widely used in performance-sensitive applications such as Firefox and Facebook. The memory is tracked by checking the total memory allocated by the program through jemalloc at a certain interval. For this paper, the interval is set at 50 milliseconds; however, in the benchmark, the interval can be set to any desired amount.

Since memory management is inherently challenging in concurrency, especially in lock-free concurrency, memory tracking serves as a highly relevant tool for debugging. We experienced this ourselves first-hand while implementing an advanced lock-free concurrent queue, where we initially encountered issues with memory management that were not immediately apparent. Because our implementation compiled and ran without errors, it was only via memory tracking that we identified inefficiencies in our memory (memory leak). This test allowed us to analyse the memory in real time, revealing behaviours that otherwise probably would have gone unnoticed. By utilising the memory tracking, we were able to debug our implementation and make sure our lock-free implementation operated as intended.

4.1.5 Ordering

Furthermore, an important and interesting test involves evaluating whether a queue maintains its original order after executing operations in a certain predefined order. This helps test if a queue exhibits relaxed properties or not, meaning it may prioritise performance optimisation over strict ordering preservation [7]. In other words, does the queue occasionally dequeue items out of order to enhance execution speed? This test is crucial to determine whether a certain data structure is suited for workloads where order integrity is critical versus when throughput and performance take priority. This is especially important given that many Rust ecosystem queues are not explicit about their ordering semantics.

This was implemented into the framework as an optional test that can be added to a queue when integrated into the framework. It works by spawning several threads that enqueue items, and spawning one thread that dequeues items. Each enqueueing thread acquires a lock for a list of items that are to be enqueued, then removes the first item from the list and enqueues it, then releases the lock. This way, the order in which the items are enqueued is known. The dequeueing thread compares all dequeued items to the correct expected item (possible since the correct order is known) and fails the test otherwise.

An important distinction to make is that if the test fails, we can definitively conclude that the queue sometimes dequeues items out of order. However, the queue passing the test does not mean it will maintain proper ordering for all cases. This is a naive test, but a more rigorous test is out of scope for this thesis. The test is, however, enough to disprove the correctness of several queues.

4.2 Implemented benchmarks

This section provides information about the exact benchmarks that are implemented in the benchmarking framework. Each benchmark is designed to evaluate different performance aspects of concurrent data structures under various workloads. The selection of which benchmarks to include was done by looking at the benchmarks used by papers such as [6], [7], [12], [14], [21].

Three benchmarks were implemented: Producer-Consumer, Enqueue-Dequeue and BFS. If any thread crashes during the execution of Producer-Consumer or Enqueue-Dequeue, the benchmark is aborted, and the results are padded with zero values to maintain consistency (a queue having zero throughput means it is non-operational). Padding with zeroes in this case means setting data points to zero values, ensuring the data remains complete and uniform in structure. This approach ensures that results remain representable across all specified iterations, even when a thread crashes during execution.

4.2.1 Producer-Consumer

The first benchmark implemented was the Producer-Consumer benchmark, where mainly how efficiently the concurrent data structure handles operations under different loads is benchmarked. In this benchmark, primarily throughput is measured, defined as the number of successful operations (enqueues and dequeues) per second, and fairness, which reflects how evenly the workload is distributed across the different threads. It is optional to include empty dequeues in the throughput measurement. All configurable values specific to this benchmark can be seen in Table 4.1.

First, the queue is optionally pre-filled with a number of items before the benchmark is started. The benchmark works by spawning producer and consumer threads. All spawned threads are synchronised to begin producing and consuming simultaneously. Producers repeatedly enqueue values into the queue while consumers attempt to dequeue values from the queue. After each operation, a local variable is incremented by one to keep track of the thread's current operation count. Between every operation, several floating-point numbers are generated pseudo-randomly to simulate real-world workloads more accurately, as done in several other papers [6], [14], [15]. The benchmark runs for a certain amount of time, and when this time is up, all threads add their local operation counter to an atomic variable. The throughput is then calculated by dividing the total operations by the number of seconds the benchmarks ran. In addition, each thread's local operation count is saved in a list, which is then used to calculate the fairness as described in Section 4.1.3.

Flag	Description
`producers`	The number of producers to spawn.
`consumers`	The number of consumers to spawn.
`prefill-amount`	How many items to pre-fill the queue with.
`time-limit`	How long to run the benchmark.
`delay`	How many floating point numbers to generate.
`iteration`	How many iterations to run the benchmark.
`empty-pops`	Whether empty dequeues should be counted in throughput calculations.

Table 4.1: The configurable options for the Producer-Consumer benchmark.

4.2.2 Enqueue-Dequeue

The second benchmark implemented, called Enqueue-Dequeue, is a variation of the Producer-Consumer benchmark, with the key difference being that threads are not fixed as either producer or consumer. All configurable values specific to this benchmark can be seen in Table 4.2.

First, the queue is optionally pre-filled with a number of items before the benchmark is started. Then, the threads are spawned, which are synchronised to begin enqueueing/dequeueing simultaneously. Each thread alternates between enqueueing and dequeueing by generating a pseudo-random floating point number $r \in [0, 1)$ prior to each operation. A thread enqueues if $r > \tau$ and dequeues otherwise. After each operation, a local variable is incremented by one to keep track of the thread's current operation count. Between every operation, several floating-point numbers are generated pseudo-randomly to simulate real-world workloads more accurately, as done in several other papers [6], [14], [15]. The benchmark runs for a certain amount of time, and when this time is up, all threads add their local operation counter to an atomic variable. The throughput is then calculated by dividing the total operations by the number of seconds the benchmarks ran. In addition, each thread's local operation count is saved in a list, which is then used to calculate the fairness as described in Section 4.1.3.

Flag	Description
`thread-count`	The number of threads to spawn.
`spread`	The value of τ.
`prefill-amount`	How many items to pre-fill the queue with.
`time-limit`	How long to run the benchmark.
`delay`	How many floating point numbers to generate.
`iteration`	How many iterations to run the benchmark.
`empty-pops`	Whether empty dequeues should be counted in throughput calculations.

Table 4.2: The configurable options for the Enqueue-Dequeue benchmark.

4.2.3 BFS

Finally, the last benchmark implemented was the Breadth-First Search (BFS) [58], aimed at evaluating both the performance and correctness of the concurrent data structure in the context of graph traversal. All configurable values specific to this benchmark can be seen in Table 4.3.

The benchmark begins by loading in a graph in the form of an .mtx (Matrix Market) file. The start node for the BFS is chosen by finding the node with the largest number of edges. The BFS computes the shortest path distance from the start node to every reachable node in the graph. A sequential BFS is then executed to establish a reference point for later verification. The loading of the graph and the sequential traversal are only performed once, minimising overheads. Sequential traversal is never done in case `no-verify` is specified (see Table 4.3). Afterwards, the benchmark can run multiple iterations on a parallel version of BFS. During these runs, the time required to complete the traversal is measured, allowing for comparisons across different implementations or configurations. After each parallel execution, the resulting traversal output is compared against the previously computed sequential result to verify correctness.

Flag	Description
`thread-count`	The number of threads to spawn.
`graph-file`	Which graph file to load.
`no-verify`	Should the benchmark not verify the parallel solution against a sequential solution.
`iteration`	How many iterations to run the benchmark.

Table 4.3: The configurable options for the BFS benchmark.

4.3 Implementation of Benchmarking Framework

To adhere to the properties discussed by Kistowski et al. [30] mentioned in Section 3.2, multiple design choices were made to try and fit all of their criteria as well as possible. Their paper was selected as a foundation due to their comprehensive and wide research for benchmarking methodology, which synthesises best practices across multiple influential studies.

4.3.1 Relevance

The benchmarking framework was developed in response to the lack of nuanced comparisons and recommendations in the Rust ecosystem (see Section 1). While Rust provides a rich set of easy-to-use crates, there is currently no comprehensive benchmark that ensures that the one you need is the best one for you. Thus, this project is highly relevant to make it easier for the individual developers to optimise their workflow and for the entire Rust ecosystem as a whole.

To ensure relevance even in the long term, we designed the framework with a focus on extensions. The framework was built so that new queues and benchmarks can be added with minimal effort. To benchmark a queue in our framework, it needs to satisfy a few requirements. The queue must implement the trait[1] `ConcurrentQueue`, which can be seen in Listing 2. It contains several functions, but the most important one is `register()`, which returns a `Handle` containing a reference to the queue. After the queue has implemented the trait, it is as simple as just adding the queue to the Cargo.toml file and add the queue as a feature in the queues.rs file. Examples of this and an example of trait implementation can be found in Appendix B. The file structure for the framework can be found in Appendix A.

The ease of integrating new queues into the framework was evaluated by trying to integrate already existing crates on crates.io and developing our own queue implementations. We started with some simple lock-based queues. Afterwards, we tried implementing more complicated state-of-the-art lock-free data structures. We also used FFI to integrate C/C++ queues.

```
1  pub trait ConcurrentQueue<T> {
2      /// Returns a handle to the queue.
3      fn register(&self) -> impl Handle<T>;
4      /// Returns the name of the queue.
5      fn get_id(&self) -> String;
6      /// Used to create a new queue.
7      /// `size` is discarded for unbounded queues.
8      fn new(size: usize) -> Self;
9  }
10
11 pub trait Handle<T> {
12     /// Pushes an item to the queue.
13     /// If it fails, it returns the item pushed.
14     fn push(&mut self, item: T) -> Result<(), T>;
15     /// Pops an item from the queue.
16     fn pop(&mut self) -> Option<T>;
17 }
```

Listing 2: The traits necessary to add a queue to the benchmarking framework.

4.3.2 Reproducibility

To ensure reproducibility, our benchmarking framework was designed to produce consistent results when running the same benchmark configuration multiple times. This consistency was achieved by deliberately focusing on simple, fundamental workloads that minimise variation between benchmark runs. Additionally, we provide configurable workloads, such as queue sizes, producer-consumer ratios and operation delays, thus allowing users to execute tests under identical conditions with precise control to confirm the overall result of the data structure.

[1] A trait is similar to interfaces in languages like Java or C#.

To track changes and validate past results, detailed logs for each benchmark are maintained, including the exact system specifications and configurations used. This allows the users to compare results across different machines as well. However, due to the inherent nature of concurrency, it is challenging to consistently get the same results. While this is a challenge, via the design choices mentioned above, it is possible to get very similar results, which shows the reproducibility factor of the framework. By integrating these practices during the development of the framework, it was ensured that anyone using the framework can achieve reliable and repeatable results.

4.3.3 Fairness in benchmarking

The fairness of the benchmarking framework was maintained by ensuring that all tested data structures are evaluated under the same, identical conditions, without favouring any particular approach or implementation. The `ConcurrentQueue` trait was designed not to favour any queue, and the benchmarks only evaluate very basic and fundamental workloads. It is also possible to vary the workloads, in case some workloads favour some queues over others. Through these measures, we ensured that differences in performance that may arise come from the data structures themselves rather than inconsistencies in the test environment.

4.3.4 Verifiability

To uphold verifiability, we decided to make the entire project open source, meaning that all of the code would be publicly available for everyone to view, modify or distribute[2]. By making the project open source, transparency and inclusivity were promoted, allowing anyone to see how the code works and how the measured data is obtained. Additionally, we wrote an extensive user guide on the GitHub page to explain how the benchmarking framework functions, including setup instructions, configuration options and the supported built-in data structures. This enables users to replicate tests in their user environments to verify the accuracy of our findings and compare their results with ours.

To further strengthen verifiability, we decided to incorporate C/C++ implementations using FFI in the benchmarking framework as well. These implementations have been extensively studied and have well-established and well-documented results. So, by incorporating these structures, we could see if our result aligned with their proven result. If our findings match their proven, established results, this further enhances the verifiability of our benchmarking framework.

4.3.5 Usability

By testing other benchmarking frameworks such as the Problem-Based Benchmarking Suite (PBBS) [43], we immediately realised the importance of usability in the project, to make the benchmarking framework as easily accessible as possible to a broad audience. Many other benchmarking frameworks require the user to navigate somewhat

[2] Link to GitHub can be found here.

challenging configurations with limited documentation, leading to uncomfortable usability.

To address this, we have made several design choices aimed at enhancing the user experience. Firstly, we have an extensive user guide on the GitHub page to explain how the benchmarking framework functions, including setup instructions, configuration options and the supported built-in data structures. Through this, it is simple to run a benchmark, even for users with no experience with the benchmarking framework. Additionally, an emphasis was placed on code readability and documentation with comments in the code itself to enhance understandability for the users who might want to extend or modify the framework.

Furthermore, to enhance usability, the benchmarking framework supports real-time logging with five distinct levels of verbosity, ranked from most critical to least critical:

1. `error`
2. `warn`
3. `info`
4. `debug`
5. `trace`

Users can control the logging level by changing the level of the environment variable `RUST_LOG`, allowing the user to customise the amount of information they receive from the benchmarking framework according to their needs. Through a focus on good usability, it was ensured that the framework would be not only powerful but also practical and easy to use for a wide range of users.

4.4 Integration of queues into framework

To ensure the benchmark framework was tested with a diverse set of concurrent queues, several research papers and crates.io were explored for relevant implementations. The selection process was based on two main criteria:

1. Well-known, famous algorithms
2. By popularity

Firstly, well-known, famous algorithms, such as Linked Concurrent Ring Queue (LCRQ) [14] or the Michael-Scott Queue [13], because they have been widely studied in literature and serve as a good reference point for new implementations. Secondly, by popularity, to reflect real-world usage of concurrent queues in the Rust ecosystem. Implementations were selected based on their popularity, measured by download count, community engagement and maintenance status on crates.io. By including widely used crates such as implementations from crossbeam and concurrent-queue, it was ensured that our results are relevant to the average developer using concurrency in Rust.

By combining well-known, famous algorithms with practical, popular ones, a set of queue implementations that provided a comprehensive performance comparison was

collected. However, it was quickly realised that the concurrent domain was not quite fully developed and was still evolving, so after popularity was considered, the options that could be found matching the focus area were chosen. This approach ensured that the benchmarking framework remains useful for both the average developer using concurrency and researchers studying concurrency.

All the queues integrated into the benchmarking framework from crates.io can be found in Table 4.4. Some queues required more work than others to add to the benchmarking framework. Wfqueue required us to fork the original repository and make some changes to the code for it to compile[3], while Bbq required us to use a less stable version of Rust, with newer features [59].

Name	**Crate**	**Download Count**[4]
ArrayQueue	Crossbeam	50 022 591
SegQueue	Crossbeam	50 022 591
atomic_queue::Queue	atomic-queue	12 136
Bounded	concurrent-queue	94 525 232
Unbounded	concurrent-queue	94 525 232
lf_queue::Queue	lf-queue	1 215
lockfree::Queue	lockfree	772 565
lockfree::Stack	lockfree	772 565
scc::Queue	scc	10 279 854
scc::Stack	scc	10 279 854
scc2::Queue	scc2	1 346
scc2::Stack	scc2	1 346
Wfqueue	wfqueue	3 760
Bbq	bbq-rs	2 022

Table 4.4: Queues included in our benchmarking framework from crates.io.

Since the benchmarking framework supports the integration of C and C++ queue implementations, a few of these were also included to provide a broader comparison, and since some research queues could not be found in the Rust ecosystem. These queues can be seen in Table 4.5.

[3]Code for fixed Wfqueue can be found here.

[4]The download count corresponds to the parent crate, not the specific queue implementation.

Name
Boost [60]
moodycamel [61]
LCRQ [14]
LPRQ [15]
FAAArrayQueue [40]

Table 4.5: C/C++ queues included in the benchmarking framework.

4.4.1 Creation of Rust queues

We implemented several lock-free FIFO queues ourselves in Rust, partly to evaluate the framework and partly to evaluate the process of creating lock-free queues in Rust. Table 4.6 shows a list of all the queues implemented.

Name	Characteristics
Bounded Ringbuffer	A simple lock-based bounded FIFO queue, implemented in a ring structure.
BasicQueue	A simple lock-based unbounded FIFO queue, implemented using a list.
MSQueue	A lock-free unbounded FIFO queue. Based on the Michael-Scott Queue [13].
TsigasZhang	A scalable lock-free concurrent FIFO queue without memory management. Based on [17].
TsigasZhang (Epoch)	A scalable lock-free concurrent FIFO queue with memory management via epoch-based reclamation. Based on [17].
TsigasZhang (HP)	A scalable lock-free concurrent FIFO queue with memory management via hazard pointers. Based on [17].
LCRQ	A lock-free unbounded queue for x86 processors with memory management via hazard pointers. Based on [14].
LPRQ	A portable lock-free unbounded queue with memory management via hazard pointers. Based on [15].
FAAArrayQueue	A portable lock-free unbounded queue with memory management via hazard pointers. Based on [40].

Table 4.6: Our own queue implementations included in the benchmarking framework.

All the lock-free queues we implemented were based on proven queues from existing papers, all containing pseudo-code showing how to implement the queues. This pseudo-code was almost always written as C/C++ like code, and the first task of implementing any of the queues was to translate that to Rust code.

Many of these queues make use of pointers and null values, something that is generally avoided when writing Rust, as they can lead to undefined behaviour. We tried, most of the time, to avoid using pointers and especially null values to make use of Rust's safety features. One recurring challenge was handling queue values

that could be null, populated, or exist in multiple distinct states. To address this, we implemented enumerators (as shown in Listing 3). This approach enhances both code safety and readability by enforcing type constraints and explicitly defining possible states. However, as Section 6.4.1 will discuss, this type-safe implementation incurs measurable performance penalties compared to direct pointer manipulation, creating a trade-off between safety and efficiency.

Despite our efforts to utilise Rust's safety guarantees, implementing these concurrent queues required using unsafe code. Rust encourages thinking about data in terms of exclusive ownership or shared immutable references, but lock-free algorithms often operate on a more granular level, with thread safety emerging from carefully orchestrated sequences of atomic operations rather than from compile-time ownership rules. While it might be theoretically possible to implement these queues using completely safe code, we were unable to. To easily be able to do CAS on more abstract types, like an enumerator or a structure, we made use of atomic pointers. This means having to use unsafe code to then read the value of the pointers. It also introduces problems related to memory management, as Rust will not free the memory that is pointed to unless specifically told to.

The memory management was one of the more difficult aspects of implementing the queues. Partly because it is difficult to manage the memory of a concurrent lock-free data structure, and partly because of Rust. Many of the papers gave some indication of how to handle the memory; however would often omit details to keep their pseudocode cleaner looking. These omissions occasionally led to memory leaks during the development of our implementations.

Given Rust's ownership model and our use of atomic pointers, we frequently needed to allocate data on the heap using `Box::new()` and convert it to raw pointers with `Box::into_raw()`, as shown in Listing 4. This conversion transfers memory management responsibility from Rust's ownership system to the developer, requiring careful attention to memory deallocation. This meant we had to be very careful to free memory that was placed on the heap. This, in conjunction with the fact that Rust does not free what atomic pointers are referencing, resulted in us having to implement the `Drop` trait for the queues. That involves defining a function that runs before Rust frees the memory of your type, where you can manually free the memory of things not freed by Rust. Implementing this was non-trivial and often involved writing unsafe code.

To make sure that our implementations were as generic as possible, we would utilise `MaybeUninit<T>`. `MaybeUninit<T>` is a wrapper type that allows the user to construct uninitialised instances of the type `T`. Most importantly, it would allow us not to force the user to have a copyable type inside their queue, a desirable trait for most generic data structure implementations in Rust. Usage of `MaybeUninit<T>` is relatively simple for the most part, however, we would constantly face issues when trying to dequeue an item. The main issues we encountered during dequeue operations were safely reading the value from `MaybeUninit<T>` without creating a double ownership scenario, ensuring that we do not free the value twice (once when reading it, and once when the queue itself is freed), and properly maintaining the internal state of the queue

after an item is removed. We found this part of creating the queues not inherently easy to grasp, as well as confusing to work with.

```
enum CellValue<E> {
    Empty,
    ThreadToken(usize),
    Value(MaybeUninit<E>),
}
```

Listing 3: Enumerator from our LPRQ implementation.

We developed these queue implementations within our benchmarking framework from the beginning, which significantly streamlined our workflow. This approach allowed us to initially focus on basic enqueue and dequeue functionality using test modules within each queue's module, which could be evaluated independently of the framework. Once these basic operations were functional, we gained immediate access to the framework's comprehensive testing capabilities. This enabled us to benchmark our queues under various conditions without writing manual tests, helping us identify numerous bugs during development. Furthermore, we could instantly compare our new implementations against existing queues to verify expected performance and monitor memory usage to detect leaks. Optimisations to the queues could be measured, as benchmarking consistently throughout the development was possible.

```
let new_val = Box::into_raw(Box::new(CellValue::Empty));
if cas2_w(
    node,
    create_safe_idx(safe, h),
    val,
    create_safe_idx(false, h + RING_SIZE as u64),
    new_val)
    {
    unsafe {
        let boxs = Box::from_raw(val);
        if let CellValue::Value(r_val) = *boxs {
            return Some(r_val.assume_init());
        }
    }
} else { unsafe { drop(Box::from_raw(new_val)); } }
```

Listing 4: Code from the LCRQ implementation showing freeing of memory.

4.5 Specifications of test bed

To ensure we have as consistent results as possible, we ran all of our benchmarks on the same dedicated server provided to us by Chalmers University of Technology. By doing this, we minimise external factors such as hardware variability, background processes and resource contention, thus ensuring that all performance differences were due to the data structures and not some other implication from the hardware/software used.

The server provided had these specifications: Two physical Intel Xeon E5-2695 v4 CPUs with a clock speed of 2.1 GHz but can be boosted to 3.3 GHz, where one processor has 18 physical cores but due to hyper-threading, it has 36 logical cores. So in total 36 physical cores and 72 logical cores where all even numbered cores is in processor one, while all the odd ones are in processor two. The Cache architecture is as follows. The L1 cache is 1.1 MiB split across all 36 cores and the L2 cache is 9 MiB also split all 36 cores and finally the L3 cache is 90 MiB, shared between every core across the two processors.

4.6 Licence

As previously mentioned in Section 4.3.4, the benchmarking framework is open source. The open source license that fits our needs the most is the GNU General Public License version 3 (or GPLv3 for short) [62]. This guarantees end users the freedom to run, study, share, or modify the software. The GPL is also a copyleft license, meaning that all derivative works must be distributed under the same or equivalent terms. This is relevant since all work that uses our project should also be distributed under the same terms, being open to all, to further enhance the ecosystem.

5

Results

This chapter presents results gained by benchmarking queues in the framework. The queues were first benchmarked using simple workloads to evaluate their raw performance characteristics, and subsequently benchmarked within the context of a real-world application, specifically, a BFS.

To ensure reliable results, each data point in the presented graphs is a mean of ten independent test runs, as done in [14], [21]. This averaging helps mitigate fluctuations and outliers to provide a clearer picture of the overall trends. In addition, all benchmarks are performed on one processor. For throughput and fairness plots, higher is better, while for mean peak memory usage and BFS plots, lower is better.

5.1 Performance, fairness and memory

The queues were benchmarked on the Enqueue-Dequeue benchmark (see Section 4.2.2) and executed for one second, with no pre-fill, not counting empty dequeues in the throughput, and 10 floating point numbers generated between each operation. Due to space constraints and the fact that the Producer-Consumer benchmarks had very similar results as the Enqueue-Dequeue benchmarks, we omit those results for further clarity.

Figure 5.1 presents the throughput and fairness for the bounded queues with τ being 0.3, 0.5 and 0.7. For all three values of τ, Wfqueue outperforms all other bounded queues in the ecosystem in throughput by a significant margin. Most other bounded queues scale negatively with an increase in thread count, especially when $\tau \in \{0.5, 0.7\}$. In Figure 5.2, Wfqueue can be seen compared to the C++ versions of LCRQ, LPRQ and FAAAQueue. Notably, Wfqueue performed on par with those, which caught our attention. After subsequent analysis using the ordering test (see Section 4.1.5), it was found that the Wfqueue occasionally violates FIFO semantics by dequeueing items out of order, thus increasing the throughput [7], [63].

We can also see that almost all bounded queue's fairness scale negatively with thread count. The outliers being our simple implementations of a lock-based bounded ringbuffer and the ArrayQueue, which are the only ones that hold a somewhat consistent fairness even though the thread count increases. Notably here as well is that the previously best performing queue in throughput, that being the Wfqueue, has consistently very bad fairness.

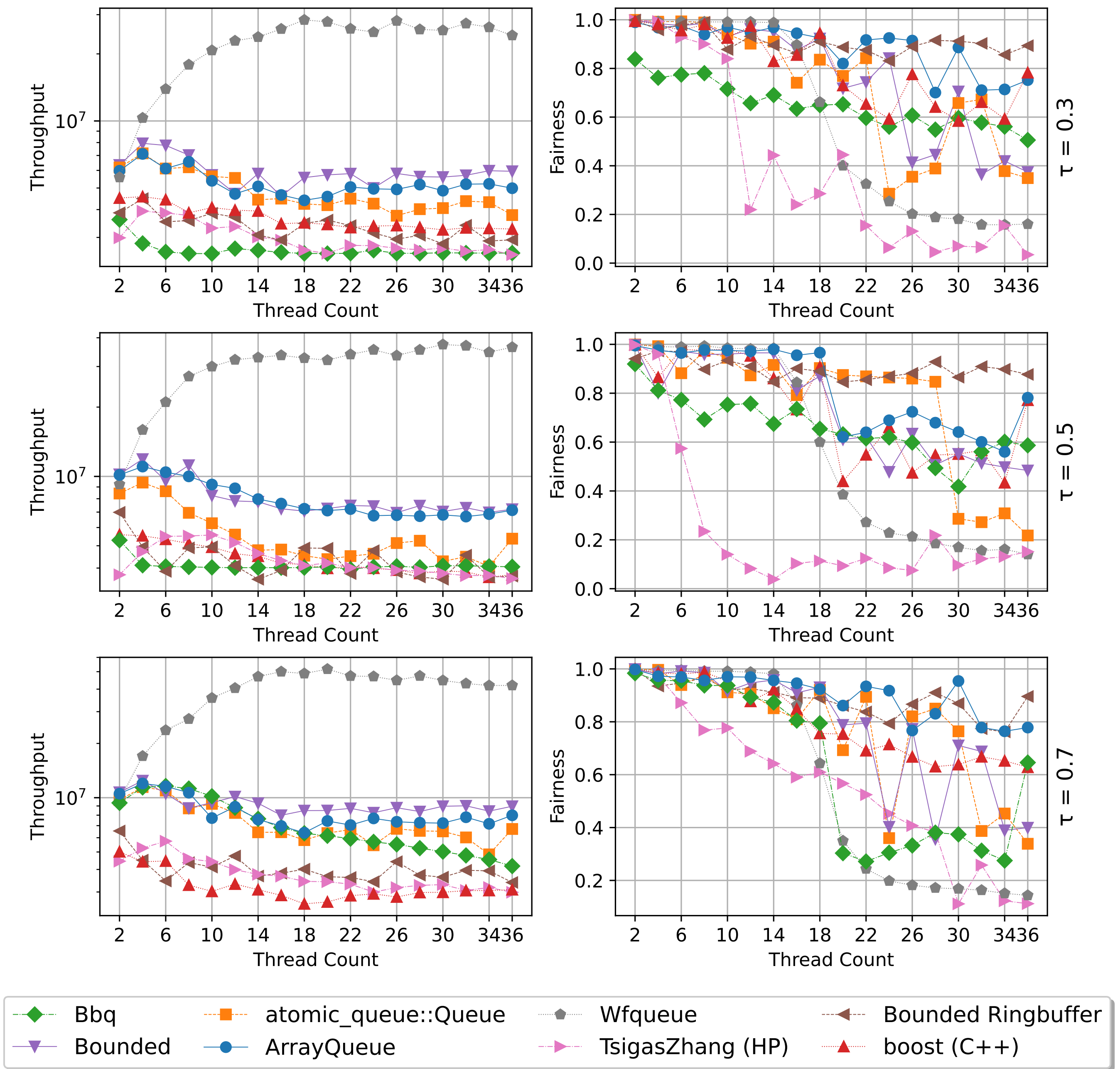


Figure 5.1: Throughput (Ops/s) and fairness comparison of bounded queues with $\tau = 0.3$, 0.5, and 0.7. For each τ value: Left: Throughput across multiple thread counts. The y-axis uses a logarithmic scale. Right: Fairness across multiple thread counts.

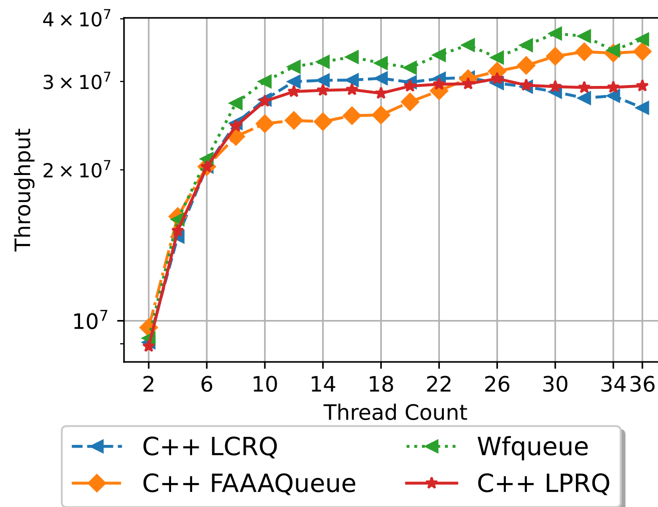


Figure 5.2: Throughput across multiple thread counts for Wfqueue against the C++ queues LPRQ, LCRQ and FAAAQueue ($\tau = 0.5$). The y-axis uses a logarithmic scale.

As for the unbounded queues, Figure 5.3 presents the throughput for both the Rust ecosystem and C++ queues, along with the queues developed in this work, with τ being 0.3, 0.5, and 0.7. Our Rust implementations of LCRQ, LPRQ and FAAAQueue outperform all Rust ecosystem queues after four threads when $\tau \in \{0.5, 0.7\}$. For $\tau = 0.3$, our Rust LPRQ and LCRQ implementations perform relatively poorly. This is probably due to not having any optimisations for when the queue is empty. However, our Rust FAAAQueue outperforms all Rust ecosystem queues at $\tau = 0.3$ as well.

The queues from SCC and SCC2, when benchmarked, consistently gave poor performance compared to almost all unbounded queues that were benchmarked. As well as that, when running almost any benchmark with more producers than consumers, another problem arose. The run time of one benchmark iteration increased from 1 second to over 40 minutes, at which point testing was terminated, suggesting the actual completion time could be substantially longer. The problem occurred with both jemalloc and the default allocator. This is why Figure 5.3 with $\tau = 0.7$ has no results from any SCC/SCC2 queues.

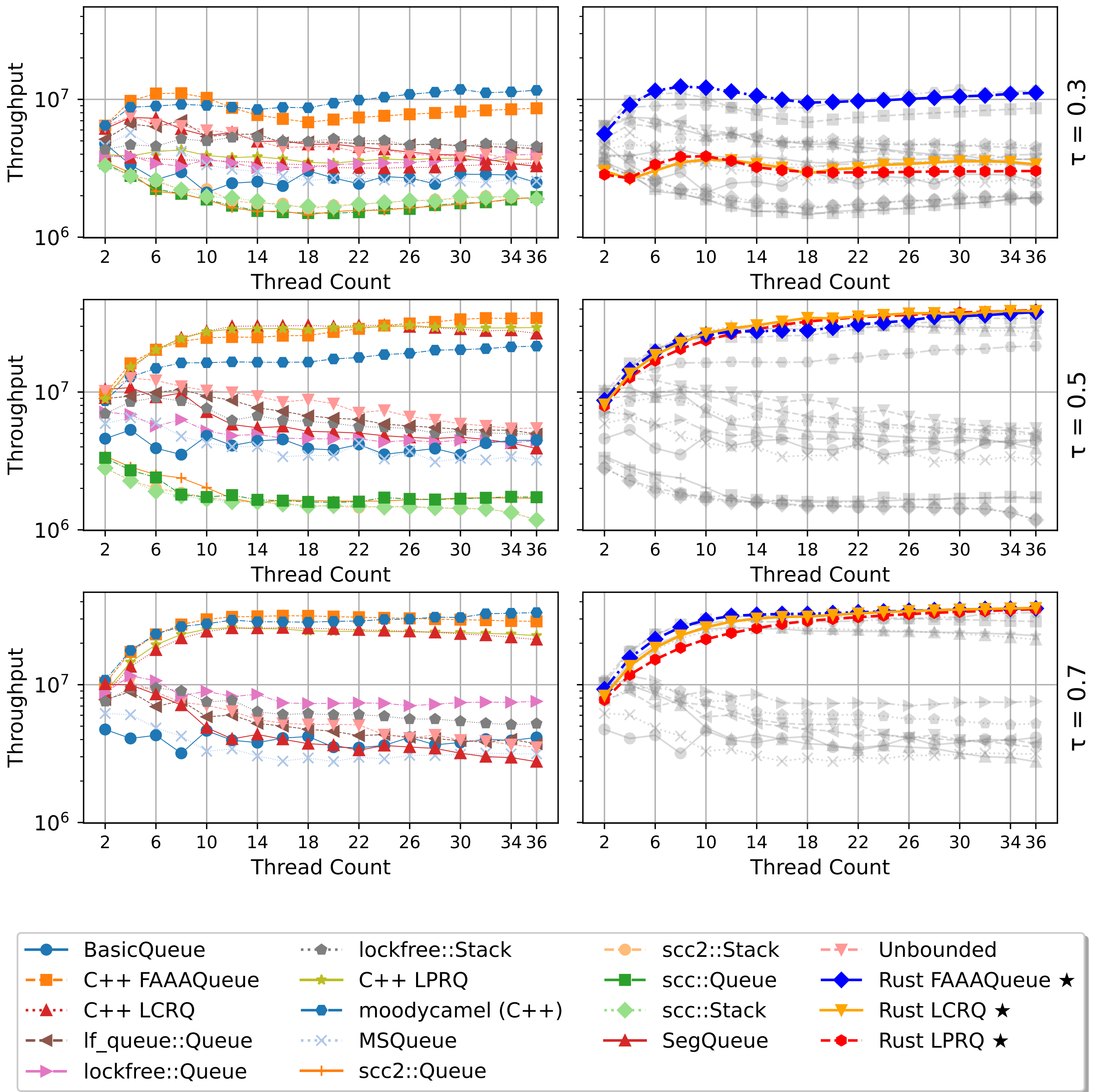


Figure 5.3: Throughput (Ops/s) comparison of unbounded queues with τ = 0.3, 0.5, and 0.7. The y-axis uses a logarithmic scale. For each τ value: Left: Existing ecosystem and C++ implementations. Right: Our Rust implementations (highlighted) shown in context with the same queues (greyed) for reference.

Figure 5.4 shows the fairness across multiple thread counts for unbounded queues of the Rust ecosystem and C++ queues compared to the queues developed by us, with τ being 0.3, 0.5, and 0.7, respectively. There is a relatively large disparity between queue fairness for all three values of τ. Our implementations of LCRQ, LPRQ and FAAAQueue manage to keep their fairness relatively high throughout the entire benchmark compared to most other queues when $\tau \in \{0.5, 0.7\}$. For $\tau = 0.3$ however, LCRQ has a drastic drop in fairness at around 8 threads, achieving poor fairness throughout the rest of that benchmark.

An interesting observation is that most of the queues exhibit a low point or a drastic drop in fairness at exactly 18 threads. This behaviour can be explained by the specifications described in Section 4.5, where 18 physical cores are available. Once this threshold is exceeded, hyper-threading is enabled, meaning that one or more cores are assigned two threads each. As a result, performance and fairness are reduced, since execution is no longer truly parallel with one thread per core.

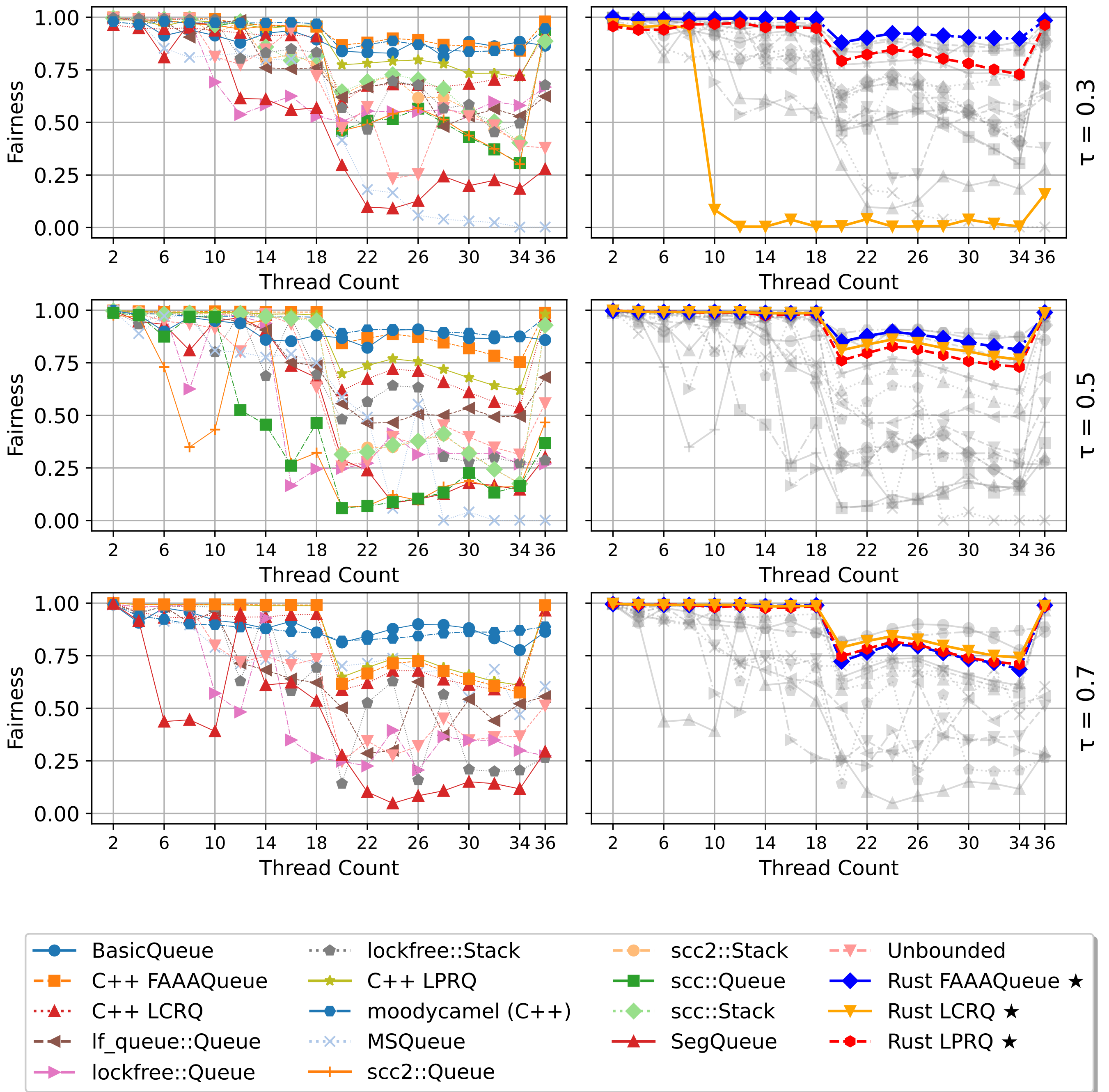


Figure 5.4: Fairness across multiple thread counts for all implemented unbounded queues with $\tau = 0.3$, 0.5, and 0.7. For each τ value: Left: Existing ecosystem and C++ implementations. Right: Our Rust implementations (highlighted) shown in context with the same queues (greyed) for reference.

Figures 5.6 and 5.5 show the average peak memory allocated from 10 runs of the Enqueue-Dequeue benchmark for unbounded and bounded queues at 36 threads with $\tau = 0.5$. There was no pre-fill, and 10 floating-point numbers were generated between each operation. Important to note is that this is not only the memory allocated by the queue, but also the memory allocated by the benchmark. However, that amount should be equal for all queues. The bounded queues were initialised with a capacity of 10^5 64-bit integers. In Figure 5.5, Bbq and the TsigasZhang queue can be seen having a significantly higher peak memory allocation compared to other bounded queues. In Figure 5.6, the queues from the crate lockfree can be seen having the highest peak memory allocation. In addition, our implementations of LCRQ, LPRQ and FAAAQueue can be seen having a much larger peak memory allocation than most of the Rust ecosystem queues.

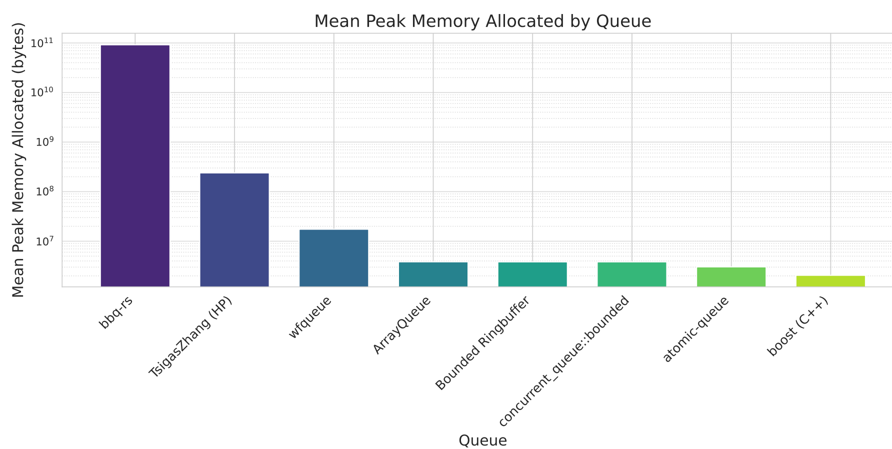


Figure 5.5: Mean peak memory usage of the bounded queues at 36 threads in the Enqueue-Dequeue benchmark ($\tau = 0.5$). The y-axis uses a logarithmic scale.

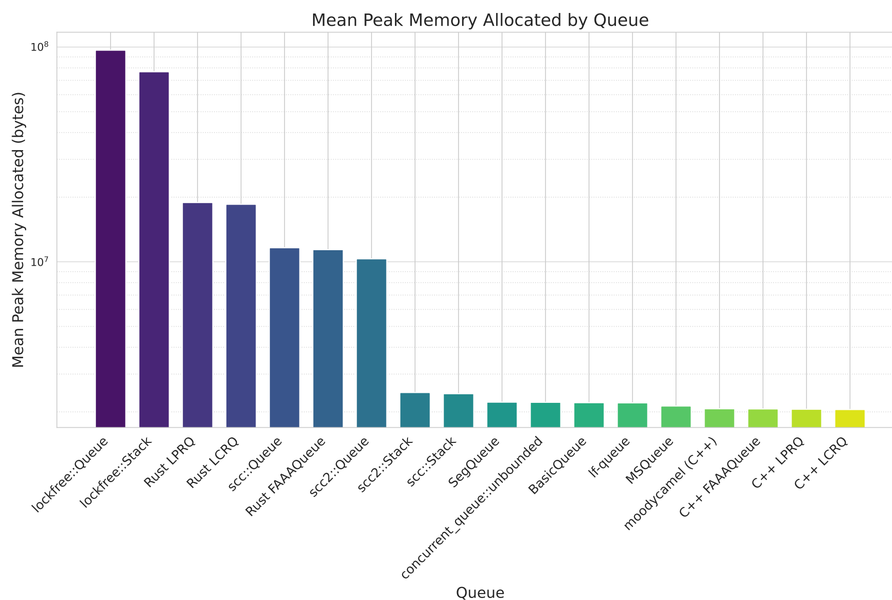


Figure 5.6: Mean peak memory usage of the unbounded queues at 36 threads in the Enqueue-Dequeue benchmark ($\tau = 0.5$). The y-axis uses a logarithmic scale.

5.1.1 lockfree::Queue crashing

Figure 5.7 presents the results from four different Enqueue-Dequeue benchmarks conducted on the FIFO queue from the lockfree crate with no pre-fill and 10 floating-point numbers generated between each operation. As the number of threads increases, all benchmarks exhibit stable performance up to a certain point, after which they begin to fail. Notably, all configurations start to crash when the thread count approaches approximately 260, indicating a potential scalability limitation or internal synchronisation bottleneck within the crate under high contention. The reason why there is some throughput even though crashes are occurring is due to the padding with zeroes described in Section 4.2. Not all iterations crash; therefore, the mean will be greater than zero.

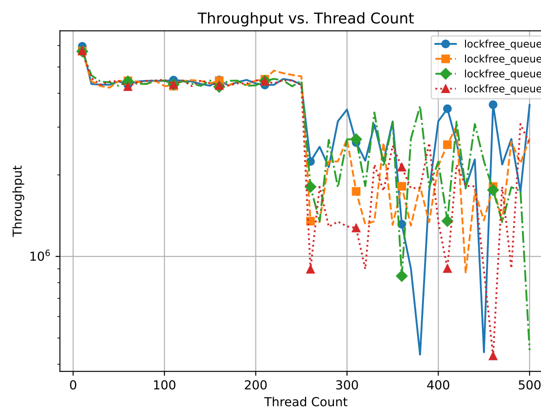


Figure 5.7: A plot of several benchmarks on the lockfree::Queue. The y-axis uses a logarithmic scale.

5.1.2 Queue optimisations

Figures 5.8 and 5.9 present the performance difference between the initial unoptimised implementation and the optimised version of our Rust implementations of LPRQ and LCRQ, respectively. We evaluated the queues on the Enqueue-Dequeue benchmark with $\tau = 0.5$, no pre-fill and 10 floating point numbers generated between each operation. The benchmark iterations were executed for one second each. The initial unoptimised versions of LCRQ and LPRQ performed relatively poorly compared to their corresponding C++ implementations, especially at higher thread counts. The optimised versions, however, surpassed their corresponding C++ versions at higher thread counts, while having relatively minimal differences in throughput at lower thread counts.

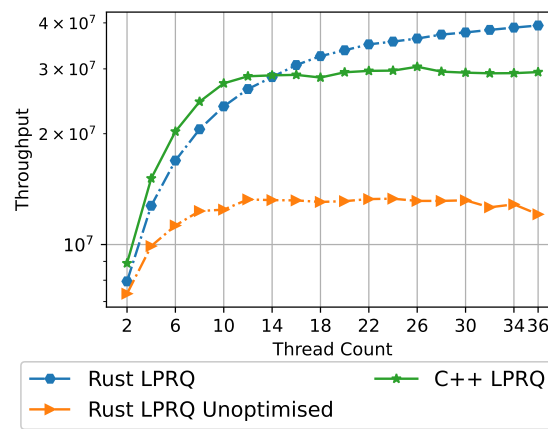


Figure 5.8: Throughput (Ops/s) of an unoptimised and an optimised version of LPRQ written in Rust and the throughput of a C++ LPRQ implementation ($\tau = 0.5$). The y-axis uses a logarithmic scale.

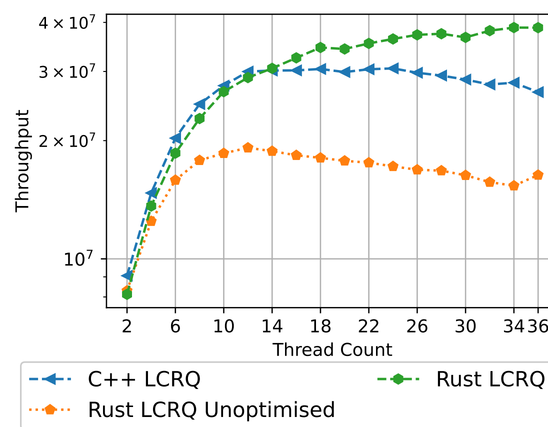


Figure 5.9: Throughput (Ops/s) of an unoptimised and an optimised version of LCRQ written in Rust and the throughput of a C++ LCRQ implementation ($\tau = 0.5$). The y-axis uses a logarithmic scale.

5.2 BFS benchmarks

Figure 5.10 shows the time for the unbounded and bounded queues to complete a BFS for two graphs. On the left is the YouTube social network, and on the right is the Twitter social network. The left graph has a graph size of 1.9 million edges and 490 thousand vertices, and the right has a graph size of 265 million edges and 21.3 million vertices. The benchmark was run 10 times with a thread count of 36, and the mean of the times is plotted. A lower value here implies the benchmark finished quickly, while a larger value means it took a longer time.

The bounded queues were initialised to a size of 10^6 64-bit integers for the soc-youtube graph, and 20×10^6 64-bit integers for the soc-twitter-2010 graph. The large sizes were chosen to minimise extra time spent due to the queue being full. Bbq crashed when trying to allocate a larger size, and thus could not be benchmarked on the soc-twitter-2010 graph.

As shown here, the SCC/SCC2 queues are the slowest, which is consistent with previous results (see Figure 5.4). On the other spectrum, our own implementations are in the lead with Rust FAAAQueue and Rust LPRQ doing exceptionally well, even outperforming their C++ counterparts.

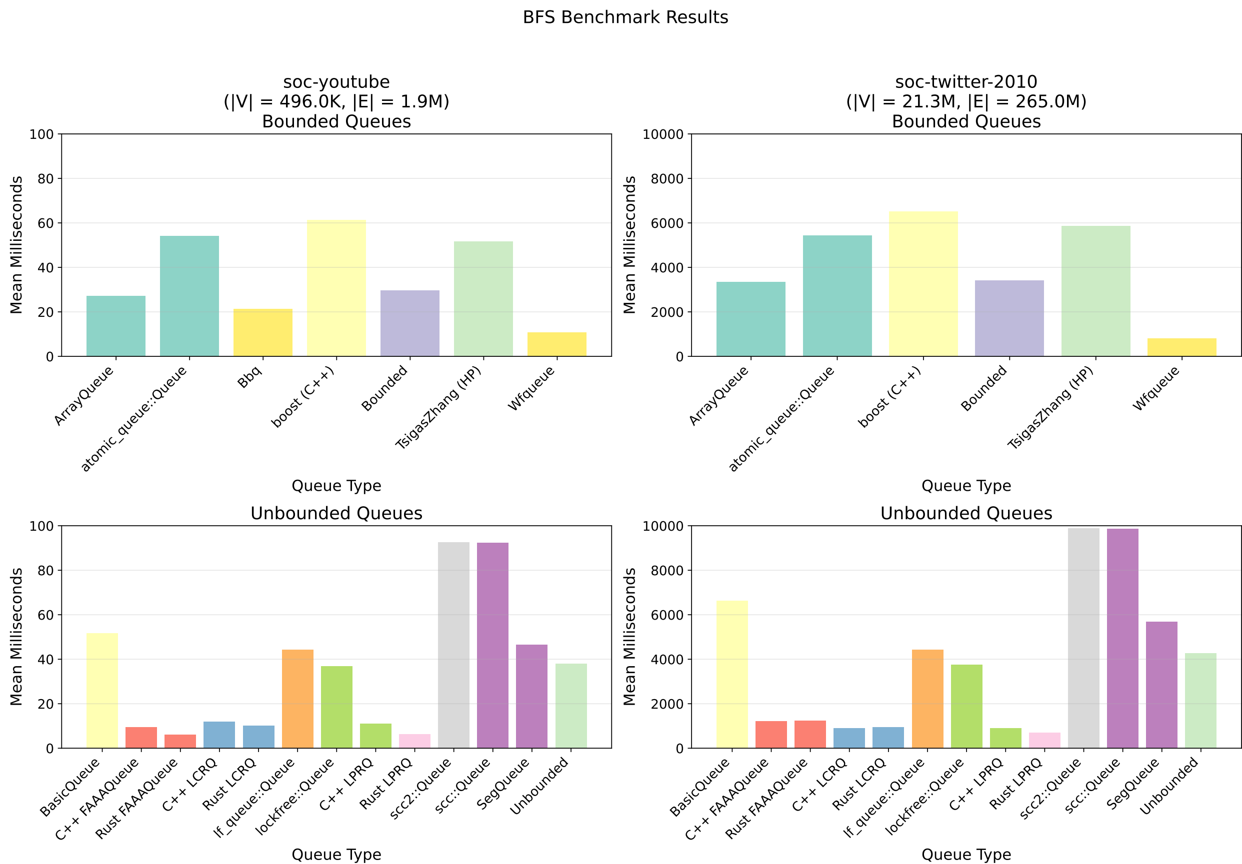


Figure 5.10: Comparing unbounded and bounded queues mean time to complete BFS. Lower is better. Graphs available at NetworkRepository [64].

6
Discussion

Here, we summarise the key findings of our benchmark study and discuss and reflect on its implications.

6.1 Results summary

This thesis aimed to develop a comprehensive benchmarking framework for concurrent queues in Rust and evaluate the performance characteristics of various implementations. Our benchmarking framework revealed several key findings. In the benchmarks, SCC/SCC2's FIFO and LIFO queues underperformed relative to other queues. It even performed worse than BasicQueue, while showing higher mean peak memory allocation than most Rust ecosystem queues. The ordering test (see Section 4.1.5) identified that Wfqueue sometimes dequeued items out of order, something not mentioned in the documentation of the queue [65]. The queues from the crate lockfree were found to have significantly higher peak memory usage than all other unbounded queues. In addition, it was also found that it consistently crashes after reaching a certain thread count.

Our Rust implementations of LCRQ and LPRQ were found to outperform all unbounded queues from the Rust ecosystem in most cases; however, not all. The Rust implementation of FAAAQueue was found to outperform all unbounded queues from the Rust ecosystem in all throughput benchmarks. In addition, our implementations significantly outperformed all Rust ecosystem queues in the BFS benchmarks. However, these high-performance implementations demonstrated significantly higher mean peak memory usage than most other unbounded queues (except those from the lockfree crate), likely due to our focus on throughput optimisation rather than memory allocation efficiency.

6.2 Benchmarking framework

The framework helped us make several important findings about Rust's concurrent ecosystem. Mainly, it allowed us to systematically evaluate the correctness of various crates (discussed further in Section 6.4). The most notable findings can be seen in Table 6.2.

The framework also proved invaluable for detecting elusive concurrency bugs. While developing concurrent queues, the Producer-Consumer and the Enqueue-Dequeue benchmarks, along with the ordering test, effectively captured early development issues, while the BFS benchmark revealed subtler defects during later stages. During LPRQ, LCRQ, and FAAAQueue development, we reached a point where implementations passed all manual tests, performed as expected in Producer-Consumer and Enqueue-Dequeue benchmarks, and demonstrated correct ordering. This falsely suggested complete implementation correctness until running the BFS benchmark with large graphs exposed remaining flaws. Thanks to this, we could find very elusive bugs, which rarely happen due to minor errors in the queue implementations. Table 6.1 shows some significant and potentially more elusive bugs we found using the benchmarking framework.

Furthermore, we recognised the necessity of testing the queues in both the `dev` and `release` profiles. This revealed critical concurrency bugs that manifested differently between profiles. Several instances occurred where concurrency issues remained dormant in the `dev` profile yet emerged in `release` due to compiler optimisations (similar to the `-O` flag in GCC). Conversely, some concurrency bugs appeared exclusively in the unoptimised `dev` profile, not appearing in the `release` profile.

As detailed in Section 4.4.1, our benchmarking framework proved exceptionally valuable during lock-free concurrent queue development. By focusing on the properties outlined in Section 4.3, the framework integrates seamlessly into queue development workflows. Its utility was particularly evident during optimisation efforts. Figures 5.8 and 5.9 demonstrate the significant performance improvements achieved between initial and final implementations of LCRQ and LPRQ. The optimised versions reached as much as a 2.5x and 3.4x increase in throughput at certain thread counts.

The following subsections will discuss some of the more critical failures in the concurrent ecosystem.

Queue	**Bug**	**Cause**
All of our queue implementations	Memory leaks.	Poor memory management.
Rust FAAAQueue	Did not perform BFS correctly.	FAAAQueue algorithm implemented incorrectly.
Rust LCRQ & Rust LPRQ	Sometimes performed BFS incorrectly on large graphs.	Minor algorithm implementation mistake.
TsigasZhang	Sometimes returned items out of order.	Wrong usage of memory orderings.
lockfree::Queue	Crashes at higher thread counts.	Wrong usage of memory orderings.
Wfqueue	Could not compile.	Use of outdated packages.

Table 6.1: Major bugs found through the use of the benchmarking framework.

Finding	Impact Area
Most lock-free bounded and unbounded queues in Rust available on crates.io scale negatively with thread count.	Scalability
Not freeing memory can potentially have a large negative impact on the throughput of a lock-free queue in Rust.	Memory Management
Wfqueue sometimes being unordered.	Correctness
Trying to avoid using null pointers can be very detrimental to performance.	Implementation Design
SCC/SCC2 show poor performance in freeing memory.	Memory Management
Bbq uses a blocking interface for a non-blocking queue.	API Design
Queues should be tested in both debug and release mode.	Queue Testing
Bbq, Lockfree queue, and SCC/SCC2 take excessive memory.	Memory Management

Table 6.2: Findings made through the use of the benchmarking framework.

6.2.1 The critical flaw of lockfree::queue

As mentioned in Section 5.1.1, the FIFO queue from the crate lockfree consistently crashes at 260 threads. This behaviour indicates some critical limitation in its implementation, likely coming from an underlying implementation error or synchronisation issue.

Utilising our benchmarking framework, we successfully reproduced the crashes and identified their root cause: a relaxed failure ordering in a CAS operation within the `try_clear_first` method on line 10 in Listing 7. After changing to a sequentially consistent ordering, the queue no longer crashed at high thread counts. However, deeper analysis revealed a concerning design flaw in the enqueue method shown in Listing 5, particularly at lines 6 and 8. Unlike the classic Michael-Scott Queue implementation, which first modifies the next pointer of the tail using CAS before updating the tail pointer, the `lockfree::Queue` reverses this sequence. It first atomically swaps the tail pointer to the new node (line 6) and only then atomically stores the new node in the previous tail's next pointer (line 8).

This swap order creates a vulnerability: between executing line 6 and line 8, the queue's integrity is compromised. Should a thread crash or face indefinite suspension during this interval, the queue would split. This is because the dequeue method (see Listing 6) retrieves the current head and passes it to `try_clear_first`, which then attempts a CAS operation on the head pointer from the current head node to the

current head node's next. Consequently, the head could eventually reference a node whose next pointer does not point to any node, despite the head not being equal to the tail, resulting in a permanently broken data structure. Which in turn means the queue `lockfree::Queue` is not in fact lock-free. In addition, since another thread's enqueue will not take effect until the previous enqueueing thread has executed line 8, the queue is not linearizable, and therefore not a correctly implemented concurrent queue.

```
1  pub fn push(&self, item: T) {
2      let node = Node::new(Removable::new(item));
3      let alloc = OwnedAlloc::new(node);
4      let node_ptr = alloc.into_raw().as_ptr();
5
6      let prev_back = self.back.swap(node_ptr, AcqRel);
7      unsafe {
8          (*prev_back).next.store(node_ptr, Release);
9      }
10 }
```

Listing 5: lockfree::Queue enqueue method.

```
1  pub fn pop(&self) -> Option<T> {
2      let pause = self.incin.inner.pause();
3      let mut front_nnptr = unsafe {
4          bypass_null(self.front.load(Relaxed))
5      };
6
7      loop {
8          match unsafe { front_nnptr.as_ref().item.take(AcqRel) } {
9              Some(val) => {
10                 unsafe { self.try_clear_first(front_nnptr, &pause) };
11                 break Some(val);
12             },
13
14             None => unsafe {
15                 front_nnptr = self.try_clear_first(front_nnptr, &pause)?;
16             },
17         }
18     }
19 }
```

Listing 6: lockfree::Queue dequeue method.

```
1  unsafe fn try_clear_first(
2      &self,
3      expected: NonNull<Node<T>>,
4      pause: &Pause<OwnedAlloc<Node<T>>>,
5  ) -> Option<NonNull<Node<T>>> {
6      let next = expected.as_ref().next.load(Acquire);
7
8      NonNull::new(next).map(|next_nnptr| {
9          let ptr = expected.as_ptr();
10         match self.front.compare_exchange(ptr, next, Relaxed, Relaxed) {
11             Ok(_) => {
12                 pause.add_to_incin(OwnedAlloc::from_raw(expected));
13                 next_nnptr
14             },
15
16             Err(found) => {
17                 bypass_null(found)
18             },
19         }
20     })
21 }
```

Listing 7: The function try_clear_first from lockfree::Queue.

6.2.2 SCC/SCC2 poor performance in freeing memory

As shown in Section 5, the queue implementation from the widely-used SCC crate (along with its fork, SCC2) demonstrated multiple concerning issues during our evaluation. Beyond simply exhibiting relatively poor performance compared to other tested implementations, these queues also had another issue in scenarios with many more producers than consumers. While benchmarks successfully completed their core operations, the program would continue running for an unexpectedly long time before finally terminating normally.

Our initial hypothesis focused on potential problems during queue destruction, and upon further investigation, we confirmed that this was indeed the case. The deallocation relies on recursive calls, which take an excessively long time to complete, as seen in the Figure 6.1. Here we can see what is called a “flamegraph” of when we ran a benchmark on the SCC queue. The horizontal axis represents execution time, and the vertical axis represents how deep in the call stack we go. The colours make no difference. In the bottom left, there are a few rectangles, which represent the time spent in the actual benchmark; the rest is just recursive calls when trying to free the memory of the queue. The core problem seems to lie in the drop function for the entries in the queue, which can be seen in Listing 8. On line 7, a call to the function `next_ptr_recursive` can be seen, which is a function that is supposed to clean up the linked list starting from the supplied head. It seems as if for a relatively large queue, this takes an inordinate amount of time.

The discovery of such issues in a widely adopted crate like SCC is particularly noteworthy. While this behaviour might not significantly impact many production use cases, especially those where queues exist for the application’s entire lifecycle or where extended shutdown periods are acceptable, it represents a potentially serious inefficiency for applications requiring responsive termination. It also raises important questions about the thoroughness of testing for concurrent data structures, even in popular libraries.

```
1  impl<T> Drop for Entry<T> {
2      #[inline]
3      fn drop(&mut self) {
4          if !self.next.is_null(Relaxed) {
5              let guard = Guard::new();
6              if let Some(next_entry) = self.next.load(Relaxed, &guard).as_ref() {
7                  next_ptr_recursive(next_entry, Relaxed, 64, &guard);
8              }
9          }
10     }
11 }
```

Listing 8: The drop function for the entries in the queue from the SCC crate.

Figure 6.1: A flamegraph where the horizontal axis represents execution time and the vertical axis indicates call stack depth. Colours are used solely for visual distinction between stack frames and carry no meaning. The graph illustrates the significant time disparity between benchmark execution and memory deallocation operations.

6.2.3 Wfqueue: Hidden order violations

The results in Section 5.1 demonstrate Wfqueue's impressive performance metrics, even reaching as high as the C++ LCRQ, LPRQ and FAAAQueue in throughput in some benchmarks. However, this study uncovered a significant undocumented limitation: Wfqueue occasionally returns items out of their insertion order. This is notably absent from the queue's crates.io page as well as its documentation. Further complicating matters, the implementation required substantial effort to utilise, as we needed to fork the original repository and change the code to make the library compilable. This suggests the project may have been abandoned or is receiving insufficient attention from its maintainer.

When selecting concurrent queue implementations for production systems, developers must carefully weigh performance advantages against both correctness guarantees and the overhead associated with adopting potentially unmaintained code. The order violation exhibited by Wfqueue might be acceptable in scenarios where approximate ordering suffices, but for applications with strict ordering dependencies, this hidden behaviour could introduce subtle and difficult-to-find bugs. These findings underscore the importance of evaluating Rust's concurrent data structures, considering not only raw performance but also correctness guarantees and the health of the supporting codebase.

6.2.4 Bbq blocks threads

A significant error was identified in Bbq's implementation, which failed to execute our Enqueue-Dequeue benchmark. Upon further investigation, we discovered a discrepancy between the implementation and the original design. While the author of the bbq-rs crate appears to have implemented the core block-based bounded queue structure as described by Wang et al. in [12], they implemented a custom blocking trait that encapsulates the entire queue, as shown in line 1 of Listing 9. This modification fundamentally alters the queue's behaviour. When attempting to dequeue from an empty queue, the implementation completely blocks without guaranteeing progress. This behaviour contradicts the original paper's design, which ensures progress under all circumstances. Therefore, this implementation is incorrect with respect to the original paper.

To correct this error, we modified the `pop()` function to eliminate the busy-waiting behaviour when encountering an empty queue, instead returning `None` immediately. The corrected implementation is presented in Listing 10. With this change, the Bbq was no longer blocking and could be benchmarked by the Enqueue-Dequeue benchmark.

```rust
impl<T> BlockingQueue for Bbq<T> {
    type Item = T;

    /// Blocking until send this item successful.
    fn push(&self, item: Self::Item) -> Result<()> {
        let mut item = item;
        loop {
            match self.enqueue(item)? {
                EnqueueState::Full(it) => item = it,
                EnqueueState::Busy(it) => item = it,
                EnqueueState::Available => return Ok(()),
            }
            // yield thread, stop wasting cpu
            sleep(Duration::from_millis(SLEEP_MILLES));
        }
    }

    /// Blocking until get a item from this queue.
    fn pop(&self) -> Result<Self::Item> {
        loop {
            if let DequeueState::Ok(item) = self.dequeue()? {
                return Ok(item);
            }
            // yield thread, stop wasting cpu
            sleep(Duration::from_millis(SLEEP_MILLES));
        }
    }
}
```

Listing 9: bbq-rs's blocking trait.

```rust
fn pop(&self) -> Option<Self::Item> {
    loop {
        match self.dequeue() {
            Ok(DequeueState::Ok(item)) => return Some(item),
            Ok(DequeueState::Empty) => return None,
            Ok(_) => {
                sleep(Duration::from_millis(SLEEP_MILLES));
                continue;
            }
            Err(_) => {
                // Optimally has real error handling.
                panic!();
            }
        }
    }
}
```

Listing 10: A solution to bbq-rs's blocking problem.

6.3 Rust's queue performance potential

As observed in the results presented in Section 5, one might initially assume that Rust performs worse than other systems programming languages such as C or C++, given that almost all Rust queues from the ecosystem perform poorly compared to the C++ queues. However, as demonstrated by our implementations of FAAAQueue, LCRQ and LPRQ in Figure 5.3, lock-free Rust queues can undoubtedly perform at competitive levels, and in certain scenarios outperform C++'s lock-free queues in the framework, provided implementations leverage well-researched algorithms and incorporate a thorough understanding of concurrent programming principles.

In the Enqueue-Dequeue benchmark, when $\tau \in \{0.5, 0.7\}$, the Rust implementations of LCRQ, LPRQ and FAAAQueue reached as high as 7x and 4.6x the throughput of the best-performing Rust ecosystem queue, respectively. With $\tau = 0.3$, FAAAQueue achieved 2.6x the throughput of the best Rust ecosystem queue. Perhaps more importantly, all three queues scaled positively with an increase in thread count most of the time, while essentially all queues from the ecosystem did not. The underlying issue, then, is not the language itself. Rather, as we have shown throughout this report, the Rust ecosystem currently suffers from a lack of high-quality, correct implementations and accessible information within its concurrent programming libraries. We therefore argue that the main limitation is not inherent to Rust as a language, but stems from the current state of the community and ecosystem surrounding it.

In the BFS benchmarks, our Rust implementations significantly outperformed all Rust ecosystem queues. Our implementations of LCRQ, LPRQ and FAAAQueue performed BFS on the soc-twitter-2010 graph 4.0x, 5.3x, and 3.0x times faster than the second fastest unbounded queue from the Rust ecosystem (`lockfree::Queue`), with similar results on the other graph. This shows that our queue implementations also perform well outside of the very simple workloads in the Enqueue-Dequeue benchmark; in other words, they can also perform very competitively in real applications. In addition, it also shows that the simple workloads in the framework can provide an effective foundation for developing and optimising concurrent queues for real-world applications.

Regarding the performance of the C++ queues, it is important to recognise that the performance difference between Rust and C++ implementations can potentially be attributed to FFI overhead. When C++ code is called from Rust, the boundary crossing introduces measurable costs. This creates performance penalties not present in pure Rust implementations, which could explain the performance advantages observed in native Rust queue implementations.

6.4 State of the Ecosystem

Throughout our project, we have continuously explored the Rust ecosystem, and we were quite shocked to find out how the state truly is. Firstly, we discovered that the current state of many community crates is concerning. Most notably, there

are no guarantees regarding the correctness, performance or even the underlying design choices of these crates. In many cases, there is no evidence that the provided implementations work as intended. No tests, no benchmarking and no documentation explaining what data structures are being used or how they behave under various conditions, as in [4], [52], [65]–[67]. This leads to a lot of serious, potential pitfalls (discussed further in Section 6.4.1), which could result in their data structures fundamentally not working as expected. Comparing this to other systems languages such as C or C++, implementations in those ecosystems are often accompanied by much clearer documentation, descriptions of underlying design choices, and in many cases, scientific papers that detail the foundations on which the implementations are built [61].

As Tyler Neely, developer of the popular Rust extension for the Flamegraph tool [68] points out:

> *"Humans are terrible at guessing about performance! Especially people who come to Rust from C and C++ will often over-optimize things in code that LLVM is able to optimize away on its own. It's always better to write Rust in a clear and obvious way, before beginning micro-optimizations, allocation-minimization, etc..."*

This observation may help explain why certain crates in the Rust ecosystem fail to function as intended. In an attempt to prematurely optimise their data structures, often without understanding how the compiler optimises code, some developers end up introducing unnecessary complexity. As a result, these implementations can become fragile, inefficient, or even fundamentally broken.

Furthermore, these crates are being widely adopted by other developers, where multiple applications currently may use these crates under the assumption that they are reliable, when in fact they do not work and could potentially introduce subtle bugs or a very dangerous security situation, with critical vulnerabilities.

6.4.1 Common Rust concurrency pitfalls

During our research throughout the Rust ecosystem, we have found multiple instances of faulty or misinterpreted implementations [4], [52], [66], [67]. The biggest pitfall almost all of the queues integrated fell into was that instead of developing a queue based on an already established algorithm, they developed their own algorithms. From what we could find, only Bbq was based on a scientific paper [12]. ArrayQueue seems to have been based on a blog post [69] presenting an algorithm for a non-lock-free bounded queue. Apart from those, no other queue sourced their algorithm that we could find, or they developed their own algorithm.

Developing lock-free concurrent queue algorithms represents a formidable challenge within computer science, as evidenced by the fact that this is an actively studied topic by researchers [12], [15]. The complexity lies in handling race conditions, memory ordering, and ensuring correctness under all possible thread interleavings. Without formal verification or proven algorithmic foundations, implementations are prone

to subtle bugs that may only manifest under specific high-contention scenarios or processor architectures.

This "ad hoc algorithm" approach we observed repeatedly over multiple codebases would consistently lead to suboptimal performance and negative scaling in throughput from an increase in thread count. Critically, contrary to potential explanations attributing these deficiencies to language constraints, our implementations of LCRQ, LPRQ and FAAAQueue, discussed in Section 6.3, conclusively demonstrate that high-performance lock-free concurrent queues with throughput comparable to optimised C++ implementations can indeed be constructed in Rust when founded upon established algorithmic principles.

By definition, a lock-free data structure must provide system-wide progress, at least one thread must always be able to complete its operation in a finite number of steps [13]. However, achieving such guarantees in practice is not as simple as it might sound. It is not as easy as just not using locks. Developers must carefully reason about all possible thread interleavings and memory reorderings, especially when using low-level atomic operations and unsafe code. For instance, in the case of the `lockfree::Queue`, the issue lies in its `enqueue()` implementation, where the tail pointer can be inadvertently lost, thus violating the very principle of lock-freedom, since some active thread will not complete its operation within a finite number of steps, regardless of other threads' actions. Similarly, the Bbq queue attempts to encapsulate its low-level concurrency primitives by layering a blocking trait interface on top of a lock-free core. While this abstraction may appear user-friendly, it inadvertently introduces blocking behaviour into what should be a non-blocking context, thereby undermining the original concurrency guarantees.

As mentioned in Section 4.4.1, when working with advanced concurrent queues that get their thread safety from carefully orchestrated sequences of atomic operations, you often have to work around Rust's safety features. This can be incredibly difficult to do correctly, especially if you have come to rely on Rust's safety features. This often leads to the developer being forced to have a more fine-grained control over the management of memory, which can lead to memory leaks. This pitfall could potentially explain the high peak memory usage exhibited by the queues we developed, as well as the queues from SCC/SCC2, lockfree and bbq-rs.

Many established lock-free algorithms depend fundamentally on entries assuming multiple states, including null placeholders. We attempted to circumvent null value usage, as this contradicts Rust's core principles. Our implementation utilised pointers to enumerators representing these possible states, exemplified in Listing 3. This approach leveraged Rust's robust safety guarantees while maintaining algorithmic correctness. However, this abstraction introduced substantial performance degradation. The implementation required continuous pointer dereferencing to determine the state of an entry, which also resulted in ownership issues and constant hassles during development. We consistently encountered this pitfall while developing the queues, as that is what feels natural when developing in Rust.

Subsequent performance analysis compelled us to abandon these enumerators in favour of direct pointer manipulation. The code in Listing 3 could be radically

simplified: `Empty` represented by null pointers, `ThreadToken` implemented via the bit manipulation functions in Listing 11, and `Value` directly stored as the raw value. This transformation yielded substantial performance improvements, at the expense of compiler-guaranteed safety guarantees. However, given that the algorithmic correctness already provides theoretical safety assurances, developers should not hesitate to avoid abstract types in favour of primitive pointer operations when implementing established lock-free data structures.

```
1 fn is_bottom<T>(value: *const T) -> bool {
2     (value as usize & 1) != 0
3 }
4
5 fn thread_local_bottom<T>(thread_id: usize) -> *mut T {
6     ((thread_id << 1) | 1) as *mut T
7 }
```

Listing 11: A ThreadToken implementation using bit manipulation of the pointer.

7

Conclusion

This thesis set out to investigate the state of Rust's concurrent ecosystem, with a particular focus on the quality and reliability of lock-free queues available in the crate ecosystem. While Rust provides strong language-level guarantees around memory safety and concurrency, advertising "fearless concurrency" as a core advantage, we found that these guarantees do not automatically extend to community-developed libraries. Our findings reveal a fragmented landscape where correctness, documentation and performance vary dramatically between crates, and where critical flaws often go undetected.

To address this, we developed a robust and extensible benchmarking framework for evaluating concurrent queue implementations in Rust. Drawing on the benchmarking principles outlined by Kistowski et al. [30] with relevance, fairness, reproducibility, verifiability, and usability, we built a benchmarking framework targeting real-world use cases. Our framework enables developers to systematically assess queues under different workloads, test for correctness violations and detect performance bottlenecks. By doing so, it helps bridge the gap between Rust's safety promises and the practical reality of concurrent programming in the ecosystem.

Furthermore, we identified throughput, fairness, memory usage and ordering guarantees as essential metrics for evaluating concurrent queue behaviour. Our framework was designed to measure these aspects under both regular and irregular parallel workloads, providing a realistic and comprehensive assessment of performance and correctness. Through targeted benchmarks, including throughput under configurable workloads, mixed role execution and correctness verification via parallel BFS, we uncovered a range of serious issues: from undocumented design flaws and inefficient memory handling to silent ordering violations and fundamental breaches of lock-freedom.

Despite these problems, we also demonstrated that, when built upon sound algorithmic foundations, Rust implementations of lock-free queues can rival or even outperform established C/C++ alternatives. This confirms that Rust is fully capable as a system language for concurrency, but that its ecosystem still requires maturing in terms of reliability and engineering discipline.

Ultimately, this thesis contributes not just a benchmarking framework, but a broader call to action for more rigorous development practices, better documentation and a more reliable foundation for designing, implementing, and evaluating concurrent data

structures in Rust. We hope that our framework becomes a foundation for future testing and that our findings help move the Rust ecosystem forward towards a more mature and trustworthy platform for high-performance concurrent programming.

7.1 Future work

As demonstrated in this paper, we have shown that Rust's concurrent ecosystem remains highly fragmented. Despite Rust's strong emphasis on safe concurrency at the language level, the surrounding ecosystem of concurrent crates lacks cohesion, consistency, and maturity in several key areas. We hope that our findings can serve as a catalyst for improving this ecosystem, encouraging efforts toward better correctness, clearer documentation, and deeper shared knowledge. With continued development in these areas, we believe Rust has the potential to be recognised not just as a safe alternative, but as a serious contender in the high-performance domain traditionally dominated by C and C++.

Furthermore, our benchmarking framework was created with the intent to evolve, ensuring that it stays a relevant asset as the Rust ecosystem grows. By making the entire project open source, we are not only providing a tool for concurrent developers but also laying the foundation for future developers to keep expanding, refining and adapting the framework where it is relevant, as new challenges and innovations emerge. It is also possible that, via selective compiling, the framework can be extended to support a broader range of data structures, making it even more versatile. Our goal is that our framework can become a central point in the Rust community. Over time, we hope future developers integrate our framework into their projects, crates and research, leveraging the benchmarks to validate new implementations, optimise existing solutions and further evolve innovation in the concurrent field. We also hope that our framework can create more transparency and shed some light on Rust's parallel landscape, making it a lot easier to understand and to empower the ecosystem.

Bibliography

[1] C. N. Steve Klabnik and w. c. f. t. R. C. Chris Krycho, Accessed: 2025-02-28, 2024. [Online]. Available: https://doc.rust-lang.org/book.

[2] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd. Addison-Wesley, 2006, ISBN: 978-0321312839.

[3] The Rust Project Developers, *FFI - The Rustonomicon*, Accessed: 19-Mar-2025, 2025. [Online]. Available: https://doc.rust-lang.org/nomicon/ffi.html.

[4] brunoczim, *Lockfree crate*, Accessed: 2025-03-14, 2019. [Online]. Available: https://crates.io/crates/lockfree.

[5] J. Abdi, G. Posluns, G. Zhang, B. Wang, and M. C. Jeffrey, "When is parallelism fearless and zero-cost with rust?" In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24, New York, NY, USA: Association for Computing Machinery, 2024, pp. 27–40.

[6] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," *SIGPLAN Not.*, vol. 51, no. 8, Feb. 2016, ISSN: 0362-1340. DOI: 10.1145/3016078.2851168. [Online]. Available: https://doi.org/10.1145/3016078.2851168.

[7] K. von Geijer, P. Tsigas, E. Johansson, and S. Hermansson, "Balanced allocations over efficient queues: A fast relaxed fifo queue," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, To appear, Mar. 2025.

[8] A. Alexandrescu and M. M. Michael, "Lock-free data structures with hazard pointers," *Dr. Dobb's Journal*, Dec. 2004, https://www.drdobbs.com/lock-free-data-structures-with-hazard-pointers/184401890.

[9] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02, Winnipeg, Manitoba, Canada: Association for Computing Machinery, 2002, pp. 73–82, ISBN: 1581135297. DOI: 10.1145/564870.564881. [Online]. Available: https://doi.org/10.1145/564870.564881.

[10] O. Sinnen, *Task Scheduling for Parallel Systems.* Hoboken, NJ: Wiley-Interscience, 2007, ISBN: 978-0-471-73576-2.

[11] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Trans. Comput. Syst.*, vol. 15, no. 2, pp. 134–165, May 1997, ISSN: 0734-2071. DOI: 10.1145/253145.253159. [Online]. Available: https://doi.org/10.1145/253145.253159.

[12] J. Wang, D. Behrens, M. Fu, *et al.*, "BBQ: A block-based bounded queue for exchanging data and profiling," in *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC '22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 249–262. [Online]. Available: `https://www.usenix.org/conference/atc22/presentation/wang-jiawei`.

[13] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA: ACM, 1996, pp. 267–275. DOI: `10.1145/248052.248106`.

[14] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, Shenzhen, China: ACM, 2013, pp. 103–112. DOI: `10.1145/2442516.2442527`.

[15] R. Romanov and N. Koval, "The state-of-the-art lcrq concurrent queue algorithm does not require cas2," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 14–26, ISBN: 9798400700156. DOI: `10.1145/3572848.3577485`. [Online]. Available: `https://doi.org/10.1145/3572848.3577485`.

[16] M. Moir and N. Shavit, "Concurrent data structures," *Handbook of Data Structures and Applications*, 2019. [Online]. Available: `https://people.csail.mit.edu/shanir/publications/concurrent-data-structures.pdf`.

[17] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '01, Crete Island, Greece: Association for Computing Machinery, 2001, pp. 134–143, ISBN: 1581134096. DOI: `10.1145/378580.378611`. [Online]. Available: `https://doi.org/10.1145/378580.378611`.

[18] N. J. Yeager and R. E. McGrath, *Web Server Technology: The Advanced Guide for World Wide Web Information Providers*. San Francisco, CA: Morgan Kaufmann, 1996, ISBN: 978-1-55860-376-9.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th. Cambridge, MA: MIT Press, 2022, ISBN: 978-0-262-54247-0.

[20] J. A. Storer, *An Introduction to Data Structures and Algorithms*. Birkhäuser, 2001, ISBN: 978-0-8176-4253-2. DOI: `10.1007/978-1-4612-0075-8`. [Online]. Available: `https://link.springer.com/book/10.1007/978-1-4612-0075-8`.

[21] P. Pirkelbauer, R. Milewicz, and J. F. Gonzalez, "A portable lock-free bounded queue," in *Algorithms and Architectures for Parallel Processing*, J. Carretero, J. Garcia-Blas, R. K. Ko, P. Mueller, and K. Nakano, Eds., Cham: Springer International Publishing, 2016, pp. 55–73, ISBN: 978-3-319-49583-5.

[22] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[23] K. Fraser, "Practical lock-freedom," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-579, Feb. 2004. DOI: `10.48456/tr-579`. [Online]. Available: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf`.

[24] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 141–150, 2012.

[25] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, ACM, 1995, pp. 214–222. DOI: `10.1145/224964.224988`. [Online]. Available: `https://dl.acm.org/doi/10.1145/224964.224988`.

[26] B. N. Bershad, "Practical considerations for non-blocking concurrent objects," in *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, Pittsburgh, PA, USA: IEEE Computer Society, May 1993, pp. 264–273.

[27] M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004. DOI: `10.1109/TPDS.2004.8`.

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

[29] 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`.

[30] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15, Austin, Texas, USA: Association for Computing Machinery, 2015, pp. 333–336, ISBN: 9781450332484. DOI: `10.1145/2668930.2688819`. [Online]. Available: `https://doi.org/10.1145/2668930.2688819`.

[31] S. Sim, S. Easterbrook, and R. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 74–83. DOI: `10.1109/ICSE.2003.1201189`.

[32] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *Principles of Distributed Systems*, E. Tovar, P. Tsigas, and H. Fouchal, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 401–414, ISBN: 978-3-540-77096-1.

[33] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," in *Distributed Computing*, R. Guerraoui, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 117–131, ISBN: 978-3-540-30186-8.

[34] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 2005, pp. 253–262.

[35] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors,"

ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 5, no. 2, pp. 164–189, 1983.

[36] E. Freudenthal and A. Gottlieb, “Process coordination with fetch-and-increment,” *ACM SIGOPS Operating Systems Review*, vol. 25, no. Special Issue, pp. 260–268, 1991.

[37] G. E. Blelloch, P. Cheng, and P. B. Gibbons, “Scalable room synchronizations,” *Theory of computing systems*, vol. 36, no. 5, pp. 397–430, 2003.

[38] R. Colvin and L. Groves, “Formal verification of an array-based nonblocking queue,” in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’05)*, 2005, pp. 507–516. DOI: `10.1109/ICECCS.2005.49`.

[39] N. Shafiei, “Non-blocking array-based algorithms for stacks and queues,” in *Distributed Computing and Networking*, V. Garg, R. Wattenhofer, and K. Kothapalli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 55–66, ISBN: 978-3-540-92295-7.

[40] P. Ramalhete, *FAAArrayQueue - MPMC Lock-Free Queue*, Accessed: 2025-04-25, 2016. [Online]. Available: `http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html`.

[41] K. Huppler, “The art of building a good benchmark,” in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–30, ISBN: 978-3-642-10424-4.

[42] J. Shun, G. E. Blelloch, J. T. Fineman, *et al.*, “Brief announcement: The problem based benchmark suite,” in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’12, New York, NY, USA: Association for Computing Machinery, 2012, pp. 68–70.

[43] D. Anderson, G. E. Blelloch, L. Dhulipala, M. Dobson, and Y. Sun, “The problem-based benchmark suite (pbbs), v2,” in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’22, New York, NY, USA: Association for Computing Machinery, 2022, pp. 445–447.

[44] A. Saligrama, A. Shen, and J. Gjengset, “A practical analysis of rust’s concurrency story,” *CoRR*, vol. abs/1904.12210, 2019. arXiv: `1904.12210`. [Online]. Available: `http://arxiv.org/abs/1904.12210`.

[45] Nikolai Vazquez, *divan: Fast and simple benchmarking for Rust projects*, Accessed: April 7, 2025, 2025. [Online]. Available: `https://github.com/nvzqz/divan`.

[46] Brook Heisler, *criterion.rs: Statistics-driven benchmarking library for Rust*, Accessed: April 7, 2025, 2024. [Online]. Available: `https://github.com/bheisler/criterion.rs`.

[47] Brook Heisler, *iai: Experimental one-shot benchmarking/profiling harness for Rust*, Accessed: April 7, 2025, 2021. [Online]. Available: `https://github.com/bheisler/iai`.

[48] Rust Community, *crates.io: The Rust Package Registry*, Accessed: March 10, 2025, 2024. [Online]. Available: `https://crates.io`.

[49] Y. Sergiy S., *Bma-benchmark*, Accessed: 2025-03-22, 2024. [Online]. Available: https://crates.io/crates/bma-benchmark.

[50] G. Kosoi, *Benchmark-rs*, Accessed: 2025-03-22, 2024. [Online]. Available: https://crates.io/crates/benchmark-rs.

[51] K. Mayes, *Queuecheck*, Accessed: 2025-03-22, 2019. [Online]. Available: https://crates.io/crates/queuecheck.

[52] ctlurself (YHM404), *Bbq-rs*, Accessed: 2025-03-19, 2023. [Online]. Available: https://crates.io/crates/bbq-rs.

[53] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco, CA: Morgan Kaufmann, 2012, ISBN: 978-0123944245.

[54] S. Eyerman, P. Michaud, and W. Rogiest, "Multiprogram throughput metrics: A systematic approach," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, pp. 1–26, 2014. DOI: 10.1145/2663346. [Online]. Available: https://dl.acm.org/doi/10.1145/2663346.

[55] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1309–1320. DOI: 10.1109/IPDPS.2013.91.

[56] M. Drozdowski, *Scheduling for parallel processing*. Springer, 2009, vol. 18.

[57] J. Evans, *Jemalloc*, Accessed: 2025-03-06, 2005. [Online]. Available: https://jemalloc.net/.

[58] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of Artificial Intelligence Tools*, A. Bundy and L. Wallen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 13–13, ISBN: 978-3-642-96868-6. DOI: 10.1007/978-3-642-96868-6_25. [Online]. Available: https://doi.org/10.1007/978-3-642-96868-6_25.

[59] C. N. Steve Klabnik and w. c. f. t. R. C. Chris Krycho, Accessed: 2025-03-19, 2024. [Online]. Available: https://doc.rust-lang.org/book/appendix-07-nightly-rust.html.

[60] Boost C++ Libraries, *Boost c++ libraries: lockfree::queue*, Accessed: 2025-05-16, 2013. [Online]. Available: https://www.boost.org/doc/libs/1_53_0/doc/html/boost/lockfree/queue.html.

[61] M. Cameron, *Moodycamel::concurrentqueue*, https://github.com/cameron314/concurrentqueue/tree/2f09da73d22a47dc8a89cdd4fc4c3bfae07f4284, Commit 2f09da7, 2013.

[62] G. Project, *Gnu general public license, version 3*, Accessed: March 2025, 2007. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.html.

[63] 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.

[64] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com.

[65] quininer, *Wfqueue*, https://crates.io/crates/wfqueue, Accessed: 2025-05-07, 2020.
[66] C. P. (wvwwvwwv), *Scc*, Accessed: 2025-03-19, 2025. [Online]. Available: https://crates.io/crates/scc.
[67] D. 4. (delta4chat), *Scc2*, Accessed: 2025-03-19, 2025. [Online]. Available: https://crates.io/crates/scc2.
[68] D. Barsky and contributors, *Flamegraph-rs: A rust implementation of brendan gregg's flamegraph visualization*, https://github.com/flamegraph-rs/flamegraph, Accessed: 2025-03-21, 2024.
[69] D. Vyukov, *Bounded mpmc queue*, https://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue, Accessed: 2025-05-13, 2021.

A

File structure

Below is the file structure of the benchmarking suite for reference.

```
% Full list of queue implementations omitted for clarity

project/
|- scripts/
|- src/
|   |- benchmarks/
|   |   |- bfs.rs
|   |   |- enq_deq.rs
|   |   |- prod_con.rs
|   |- cpp_queues/
|   |   |- boost/
|   |   |- lcrq/
|   |   |- lprq/
|   |   |- moodycamel/
|   |- queues/
|   |   |- array_queue.rs
|   |   |- atomic_queue.rs
|   |   |- ...
|   |   |- wfqueue.rs
|   |- arguments.rs
|   |- benchmarks.rs
|   |- lib.rs
|   |- main.rs
|   |- order.rs
|   |- queues.rs
|   |- traits.rs
|- Cargo.toml
|- README.md
```

B

Queue implementation example

Examples of the steps needed to add a queue to the benchmarking framework.

```
1  implement_benchmark!(
2      "basic_queue",
3      crate::queues::basic_queue::BasicQueue<usize>,
4      &bench_conf
5  );
```

Listing 12: An example of the implement_benchmark macro in lib.rs.

```
1  #[cfg(feature = "basic_queue")]
2  pub mod basic_queue;
```

Listing 13: Example of feature adding in queue.rs.

```rust
 pub struct BasicQueueHandle<'a, T> {
    queue: &'a BasicQueue<T>
}

impl<T> ConcurrentQueue<T> for BasicQueue<T> {
    fn register(&self) -> impl Handle<T> {
        BasicQueueHandle {
            queue: self,
        }
    }
    fn get_id(&self) -> String {
        String::from("basic_queue")
    }
    fn new(_size: usize) -> Self {
        BasicQueue {
            bqueue: BQueue::new()
        }
    }
}

impl<T> Handle<T> for BasicQueueHandle<'_, T> {
    fn push(&mut self, item: T) -> Result<(), T>{
        self.queue.bqueue.push(item);
        Ok(())
    }
    fn pop(&mut self) -> Option<T> {
        self.queue.bqueue.pop()
    }
}
```

Listing 14: Example of how the traits are implemented for a queue.