CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lock-Free Concurrency in Rust

Analysis of Rust for Lock-Free Programming

Master's thesis in Computer science and engineering

Lilly Jinstrand, Marcus Julin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Lock-Free Concurrency in Rust

Analysis of Rust for Lock-Free Programming

Lilly Jinstrand, Marcus Julin

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Lock-Free Concurrency in Rust
Analysis of Rust for Lock-Free Programming
Lilly Jinstrand, Marcus Julin

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

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Lock-Free Concurrency in Rust
Analysis of Rust for Lock-Free Programming
Lilly Jinstrand, Marcus Julin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Among the crucial data structures for modern high-performance systems is the fundamental concurrent lock-free FIFO queue. The programming languages selected for these systems typically prioritize speed and efficiency and consequently, these languages inherently lack safety and security in their designs. Due to these drawbacks, Rust was created to maintain the benefits associated with low-level languages while also providing safety and security. It stands as the most-loved language from 2016's Stack Overflow developer survey to the most recent one in 2023, and since 2022 it has been rated as the most wanted. However, even though a popular language, not much research has been done to investigate Rust as a choice for these concurrent high-performance systems.

In this paper, we are putting Rust up to the test, analyzing its suitability for high-performance concurrent lock-free programming. For this, we implement two concurrent lock-free FIFO queues in the Rust language, one classic and the other state-of-the-art. These are then compared against their counterparts in C or C++ using a diverse set of measurements including throughput, energy, and memory. Additionally, perhaps as important, we assess our developer experience, providing a nuanced view of Rust's ergonomics, and highlighting some differences with C++. Our comparative analysis shows that Rust is a viable option in a practical high-performance concurrent scenario. Its overall convenience and compatibility with traditional tools, combined with an ecosystem that includes a unique powerful tool exclusive to Rust, makes for a user-friendly and effective programming language and, we believe it is a candidate for high-performance concurrent lock-free programming.

Keywords: Rust, lock-free, concurrency, concurrent, FIFO queue.

Acknowledgements

We thank our supervisors Kåre von Geijer and Philippas Tsigas for their support and guidance.

Lilly Jinstrand, Marcus Julin, Gothenburg, 2024-06-15

Contents

Glossary

Atomic operation: *An operation that completes in a single step relative to other threads.*

Atomicity: *The property of an operation being indivisible; other threads see it as occurring entirely or not at all.*

Borrow: *The process of creating a reference in Rust.*

Compare-and-Swap (CAS): *An atomic operation that updates a single memory location if it matches an expected old value.*

Compare-and-Swap-2 (CAS2): *Function as a CAS on two adjacent memory locations.*

Concurrency: *The property of systems to handle multiple activities simultaneously.*

Contention: *A situation where two or more threads compete for the same resource.*

Concurrent Ring Queue (CRQ): *A node in the LCRQ where the actual values of the LCRQ are located.*

Data race: *A situation where multiple threads access the same resource concurrently and at least one of the accesses is a write.*

Deadlock: *A situation where one thread indefinitely holds a lock to a shared resource, effectively blocking all waiting threads.*

Deterministic destruction: *The property that resources' memory are reclaimed in a predictable and determined order. A property utilized by Rust.*

Epoch-based memory reclamation: *A memory reclamation technique where resources are reclaimed based on "epochs", ensuring safe memory recovery in concurrent programs.*

Fetch-and-Add (F&A): *An atomic operation that increments a variable by a given value and returns the old value.*

Garbage collector (GC): *A non-deterministic automatic memory reclamation that recovers memory occupied by resources no longer in use.*

Hazard pointers: *A memory reclamation technique designed to safely reclaim the memory of nodes in concurrent lock-free data structures.*

Hot spot: *A memory location that becomes a source of contention.*

LCRQ: *A state-of-the-art non-blocking (lock-free) linearizable concurrent FIFO queue*

Lifetime: *The scope within which a reference in Rust is valid and can be used safely.*

Lock: *Synchronization mechanism that ensures only one thread accesses a shared resource at a time.*

Lock-free: *Non-blocking synchronization mechanism that ensures at least one thread makes progress when several threads access the same resource.*

LPRQ: *Modified version of the LCRQ, claiming increased portability.*

Memory reclamation: *The process of recovering memory that is no longer in use by a program.*

Michael and Scott queue (MS queue): *The most classic concurrent lock-free queues.*

Miri (Mid-level Intermediate Representation Interpreter) *Unique tool for Rust used to detect when unsafe code fails to uphold Miri's safety requirements.*

Multi-Producer Multi-Consumer (MPMC): *Workload used to benchmark the queues. Producers threads enqueue and consumers threads dequeue.*

Pairwise: *Workload used to benchmark the queues. Each thread sequentially performs an enqueue and dequeue.*

Process: *Component in a concurrent system that is delegated specific tasks.*

Progress: *The concept of when a thread can advance in its execution.*

Reference: *Pointer with the guarantee it points to a valid type.*

Synchronization: *Refers to the process of coordinating threads in a concurrent system, ensuring all threads' view of shared resources is consistent.*

Thread: *The execution unit of a process.*

Undefined Behavior (UB): *When unsafe code fails to uphold the language safety requirements, it is classified as UB.*

List of Figures

List of Tables

1

Introduction

High-performance systems are known for leveraging concurrency to manage multiple tasks simultaneously, significantly enhancing their efficiency compared to systems that handle tasks sequentially [2]. A data structure that can utilize concurrency is the fundamental first-in-first-out (FIFO) queue, referred to as a *concurrent FIFO queue*. This structure is widely used in areas where performance is paramount, such as in real-time systems and operating systems [4, 39, 50].

Furthermore, *synchronization* in these highly concurrent systems is essential to maintain the consistency of its shared resources [29]. In other words, for such systems to operate as intended, tasks that access a shared resource, such as a concurrent queue, should all view the same content. To achieve this consistency, the system can synchronize by *locking* the queue for a single task, or implement a *lock-free* mechanism, allowing several tasks to access the queue concurrently. Although the latter approach introduces more complexity, studies show that lock-free algorithms perform better than their lock-based counterparts [29]. Additionally, a lock-free approach guarantees that some tasks will always be able to access the queue, whereas a lock could block all other tasks indefinitely [15]. In the lock-free approach, a concurrent lock-free queue can synchronize by leveraging hardware-supported *atomic* operations. Tasks executing these operations are guaranteed *atomicity*, meaning all parts of the operation are executed without any interruptions, otherwise, it is not executed at all [2]. In this project, we specifically focus on algorithms utilizing atomic operations to achieve the lock-free property for concurrent queues, called *lock-free concurrent FIFO queues*.

In concurrent high-performance environments where these queues are utilized, low-level languages like C and C++ are typically the common choices. Despite their capability to deliver high performance they are often considered unsafe and inefficient, especially in the context of concurrency programming [5, 35, 41]. Because of these drawbacks, Mozilla developed Rust [43] to provide a language that preserves the performance of a low-level language while also providing safety guarantees [21]. Despite being a relatively new language, Rust stands as the most-loved language since 2016 up to the most recent Stack Overflow developer survey in 2023, and since 2022 has been the most wanted [40]. Its increasing popularity suggests that Rust is a language for the future.

Moreover, by default, the uncompromising Rust compiler guarantees safety by enforcing strict rules, ensuring programmers write safe code. However, in some use

cases, such as when implementing lock-free data structures, these rules are too stringent and must be relaxed [3]. Rust addresses this issue by allowing programmers to explicitly declare code blocks as *unsafe*, thus bypassing the rules for that specific chunk of code. This feature gives programmers more control but also increases the responsibility to ensure safety manually. For concurrent programming, this means that while remaining in the default mode, called *safe* Rust, Rust guarantees no data races, but when using unsafe blocks it does not [11]. Although implementing lock-free data structures in Rust will require the use of unsafe code, previous research has shown it can be limited to parts where it is strictly necessary [24]. Importantly, the Rust ecosystem includes a powerful tool designed to help developers identify potential issues in unsafe code.

In this thesis, we compare our implementations of lock-free concurrent FIFO queues in Rust versus existing ones in C or C++. In more detail, serving as a proof of concept, we compare Rust and C using one of the most classic lock-free concurrent queue algorithms, the Michael and Scott Queue [29]. The more rigorous comparison is between Rust and C++. For this, we utilize a portable version of the state-of-the-art LCRQ algorithm [38]. For a robust overview of each language, we include the quantitative performance metrics such as throughput, energy consumption, memory usage, and access pattern. This variety of measurements could potentially give insight into how Rust stands against C and C++ in performance-critical environments with high levels of concurrency. Additionally, in a more qualitative assessment, we deliver our developer experience in a comprehensive evaluation of the ergonomics associated with using Rust, highlighting the advantages of choosing Rust over these traditional languages, while also addressing any drawbacks. Another important aspect we consider is how much of Rust's guarantees are sacrificed by using necessary unsafe code in our implementations.

Our study shows that Rust is a programming language that can compete with these established low-level languages. Specifically, in our closest to a real-world scenario, the state-of-the-art implementation in Rust demonstrates it can achieve identical results and even outperform C++ in some of the measurements. Moreover, even when unsafe code is unavoidable, the Rust ecosystem provides Miri, an easy-to-use tool exclusive to Rust, that assists developers in writing safe code in unsafe blocks. Furthermore, common profiling tools associated with C++ are as easy to use for Rust, thus preserving this experience for C++ developers, and making for a smooth transition. Additionally, integrated into its core, Rust includes a build tool and package manager, removing the complexity often associated with this. Ultimately, we believe Rust is a handy language with a promising future and is a viable option for high-performance concurrent lock-free programming.

1.1 Related work

Most existing research on concurrency in Rust concentrates on the language's safety features, leaving the performance aspects of Rust in concurrent applications as relatively unexplored territory. In one paper by Yu et al. (2019) [54] they perform a preliminary study on Rust's concurrency safety to improve understanding of Rust's

concurrency, aiding in the creation of better Rust software and more efficient debugging tools. Another paper by Qin et al. (2020) [34] performs an empirical study addressing various aspects of Rust programming, including what concurrency bugs Rust programmers make. They found that these concurrency bugs are caused by misunderstandings about Rust's complex principles. Lastly, in a paper by Lin et al. (2016) [24] they investigate how well Rust performs compared to C by implementing, in both languages, a high-performance garbage collector, a system that typically exhibits multiple levels of concurrency. They show that the safe domain did not impact performance significantly and that unsafe code can be minimized to parts where it is strictly required. Additionally, they conclude that their experience is a proof of concept that Rust is a suitable option for their high-performance implementation. It however stands out as the only study we found that examines Rust's performance in a concurrent context, leaving it as the sole paper that somewhat aligns with what we do in this paper.

1.2 Aim

This project aims to investigate how much performance needs to be traded for Rust's additional safety guarantees in high-performance lock-free applications. By focusing on lock-free concurrent FIFO queues, this paper conducts a comparative analysis of lock-free concurrent FIFO queues, focusing specifically on implementations in Rust versus those in C or C++. This comparison enables an assessment of how much Rust's safety features impact performance and evaluates Rust's overall suitability for high-performance concurrent programming.

1.3 Limitations

For the reference implementations in C and C++, we limit ourselves to utilizing existing solutions rather than developing our own. Additionally, complex features crucial for concurrency but not inherently tied to the functionality of queues will be leveraged from preexisting Rust libraries.

1.4 Research questions

The primary research question for this project is whether Rust is a suitable choice for high-performance concurrent lock-free programming. This includes answering how much performance is sacrificed compared to C and C++, and the development complexity introduced by Rust's strict safety rules in such demanding environments. Additionally, we examine how much of Rust's safety guarantees must be compromised through the use of unsafe code.

2

Theory

2.1 Concurrent Programming

In this thesis, the term *concurrency* refers to the ability of a system to perform multiple activities simultaneously [6]. All modern computers operate on some level of concurrency and *processes* are a vital part of modeling and controlling this concurrency. These processes are delegated different tasks for the system to work as intended. One process may wait for incoming emails, and another might run on behalf of the antivirus program, etc. For a program, a process is essentially an instance of that program in execution, including the current state of the program counter, registers, and variables. [2]. Importantly, each process can consist of multiple execution units known as *threads* [6].

A concurrent program consists of multiple threads running concurrently, operating on some *shared resource*, which is a memory location to which several threads have access. For such a program to function as intended, it must ensure a consistent view of the shared resources among participating threads. It is essential to understand the concept of synchronization and how it can utilize atomic operations to ensure consistency. Additionally, it is crucial to ensure that threads can reliably make progress in a concurrent environment [36].

2.1.1 Synchronization

In concurrent programming, *synchronization* is a fundamental concept to safely handle access to shared resources [39]. For concurrent programs, synchronization is required in its *critical sections*. These sections are parts of a program that only one thread may execute at a time [15] such as when a thread wants to change the state of some shared resource. If multiple threads are allowed to change or read a resource's state simultaneously a *data race* can occur [26], which means that at least one thread will mistakenly assume it has the correct view of the resource, leading to inconsistencies.

To avoid inconsistencies, access to a shared resource's critical section must be synchronized and can be achieved through *blocking* or *non-blocking* algorithms. A blocking algorithm can be implemented through *locks*, also called *mutexes* [2]. Locks limit the access to a critical section to one thread at a time, effectively blocking all other threads until the thread holding the lock is finished. The non-blocking approach,

which is the one relevant to this project, does not enforce this restriction. Instead, a program can synchronize by utilizing atomic operations [36].

2.1.2 Atomicity

Atomicity is a condition for maintaining consistency [36]. It guarantees that an action is executed in its entirety without interruption, eliminating the chance of other threads observing the action in an intermediate state. This safeguard maintains data consistency, even as multiple threads access and modify data concurrently. Applying this condition to concurrent objects to achieve consistency is often referred to as *linearizability*, meaning that operations from several threads on an object appear as if they were executed sequentially by a single thread.

Furthermore, atomicity is practically achieved using built-in primitives known as *atomic operations*, which are specifically designed to address synchronization challenges [31]. These are implemented at the hardware level [36], and operations executed at the hardware level are directly performed on the computer's *word-sized* memory addresses. The size varies across systems but most machines today have a word size of 32 or 64 bits. Consequently, the maximum allowed size a single integer or pointer data can occupy in the memory is determined by the word size [6]. Moreover, an atomic operation performing a read from or write to some shared resource is called *load* or *store* respectively.

Additionally, depending on the instruction set architecture for a system, different atomic operations may or may not be available. However, the two atomic operations relevant to this project *compare-and-swap* (CAS) and *fetch-and-add* (F&A) are available on the dominating (64 bit) x86-64 architecture [30].

2.1.3 Compare-and-Swap

Compare-and-swap, also known as compare and set or compare exchange, is an atomic operation that performs a conditional update on a single memory location, which could store a shared resource. The process involves three key components: a target memory location (address X), and two values, *old* and *new*. CAS compares the current value of X with *old* and if they match, indicating that X has not been altered by another thread, the operation updates X with *new*. Otherwise, the operation fails, and X remains unchanged [36].

In addition, as for the algorithm in Section 2.2.2, certain scenarios require updating more than one memory location simultaneously. This can be done through the compare-and-swap 2 (CAS2) operation, designed to atomically act on two adjacent memory locations. However, some researchers argue against the reliance on CAS2, pointing out its limited availability in most modern programming languages and some architectures, which could be seen as a disadvantage [38].

2.1.4 ABA Problem

Due to CAS being dependent on a previously stored value, it is vulnerable to the *ABA problem* [15]. This problem occurs when a thread first reads the memory location X where it wants to perform CAS, in between another thread modifies X, changing its value from A to B and then back to A again. When the first thread finally performs the CAS it will succeed because X is again A. However, the context or state associated with X might have changed, leading to incorrect program behavior. In practice, the ABA problem is typically only a concern when using dynamically allocated memory in a concurrent environment for languages without any mechanism to automatically manage its programs' memory [15], see Section 2.1.7.

2.1.5 Fetch-and-Add

This atomic operation takes a memory address and a value and adds the value to the address [15]. F&A is much simpler than CAS and is a vital component in state-of-the-art non-blocking concurrent queues because of its crucial property of always succeeding [30], in contrast to CAS which can fail due to its conditional nature.

The algorithm presented in 2.2.2 leverages the F&A property to spread threads around the queue. For example, threads that are simultaneously performing their individual enqueue operation will, first of all, perform a fast F&A on the tail index. Due to atomicity, each thread will quickly receive its index in which to perform the enqueue and can operate concurrently with the other threads. Figure 2.1 shows a simplified illustration of how this would work for three threads simultaneously performing an enqueue to a queue with current tail index n.

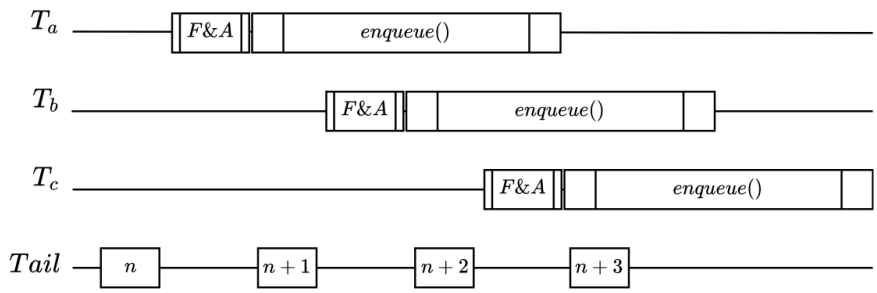

Figure 2.1: Illustration of how to leverage F&A to spread threads across a queue.

2.1.6 Lock-Free

The *lock-free* property is a non-blocking progress condition. A method is considered lock-free if it guarantees that, regularly, at least one method call is completed within a finite number of steps. This means that at least one thread will always progress when several threads access a shared resource, unlike lock-based algorithms, which could prevent progress from all threads if a single thread takes the lock and never

releases it, called *deadlock*. Although deadlock can never occur in a lock-free algorithm they risk that some threads may never make progress, however, in a practical sense, it is improbable [15]. Furthermore, achieving this property for an algorithm can be done by relying on CAS when threads need to operate on a shared resource since it is inherently lock-free; it does not allow for any thread to obstruct other threads from executing [29, 30].

Furthermore, in both lock and lock-free implementations, there is a risk of *contention* among threads in concurrent environments. This happens when several threads try to access the same shared resource simultaneously which leads them to compete in making progress. With an operation like CAS, contention can occur if several threads perform CAS simultaneously on the same resource. In such a scenario, only one thread will succeed at a time while the others fail. Due to being lock-free, failing threads are not prevented from continuously retrying their CAS, which affects the overall performance of the program [30]. Additionally, a *hot spot* is an area in the memory that becomes a source of contention [15]. Finding a solution to contention is important for performance, research even shows that enforcing a sequential execution to synchronize access to hot spots is better than contention [14]. As explained in Section 2.2.2, strategies exist to reduce the impact that contended hot spots have on performance. These methods also scale better than the sequential approach [30].

2.1.7 Memory Reclamation

A program that does not recover reusable memory will eventually run out of memory if it continuously allocates new objects to the memory. To solve this, programming languages can incorporate an automatic *garbage collector* (GC) that automatically recovers memory that is no longer needed by its programs [6]. However, this method is inherently non-deterministic which might lead to a sudden decline in performance due to a program experiencing overhead when the GC arbitrarily decides to recover memory. Due to this unexpected overhead at runtime, an automatic GC is not desirable for a high-performance environment [2]. There are however languages that rely on other automatic techniques to recover memory. Among these is Rust's deterministic approach mentioned in Section 2.3.2. Additionally, some languages such as C and C++ include neither, and the process of recovering memory must be implemented manually [39].

Furthermore, since an automatic GC is not always suitable, such as for highly concurrent environments [12], allowing programmers to implement a tailored solution manually gives more control over the performance of a program. Moreover, manually or automatically, the process of recovering memory is called *memory reclamation*, and programs without memory reclamation are said to *leak* memory until it runs out.

2.2 Concurrent Lock-Free FIFO Queues

The algorithms used in this paper are built upon the fundamental data structure *singly-linked list*. In this data structure, elements in the list are called *nodes*, and the

list supports insert and remove operations for nodes. The number of nodes decides the size, making it dynamically sized. Every node has an attribute *key* which is the value associated with the node, and a pointer attribute *next* which points to the successor node of the current node. The list itself has a pointer attribute *head* which always points to the first node in the list. By including a pointer to the last node in the list called *tail* it is possible to apply the FIFO principle, thus making it a FIFO queue. For a queue, inserting and removing nodes are respectively named *enqueue* and *dequeue* [7].

2.2.1 Michael and Scott Queue

The Michael and Scott queue, MS queue for short, is one of the oldest non-blocking concurrent FIFO queues [29]. It relies on CAS for synchronization for its enqueue and dequeue operations and utilizes a counter tag for each node to avoid the ABA problem. The algorithm presented in the paper does not explicitly handle memory reclamation which presents a problem for some languages because it is dynamically sized. However, in a subsequent paper by the same author, hazard pointers are applied to the MS queue for memory reclamation [28]. Furthermore, the reliance on CAS for synchronization leads to a significant decline in scalability, even with a low level of concurrency, due to the queue's head and tail becoming highly contended hot spots [14, 15, 30]. Figure 2.2 shows a simple MS queue.

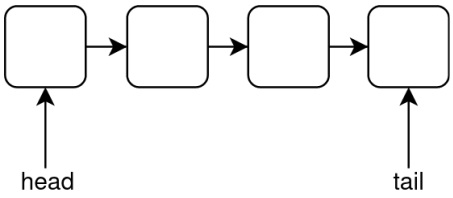

Figure 2.2: Simple MS queue with 4 nodes.

2.2.2 LCRQ

LCRQ is a non-blocking (lock-free) linearizable concurrent FIFO queue by Morrison and Afek [30]. The design of LCRQ addresses the scalability issues arising from the heavy use of CAS, such as in the MS queue. They show it is not the contended CAS hot spots in themselves that are the source of poor performance, it is the wasted work by performing unnecessary CAS operations. Instead of only relying on CAS, they circumvent the issue by utilizing F&A at the hot spots instead because of its simplicity and property of always succeeding, as mentioned in Section 2.1.5.

Fundamentally, the LCRQ algorithm is an MS queue linked list queue where each node is a concurrent ring queue (CRQ). These rings are cyclic arrays of fixed size R where most of the activity occurs for the LCRQ. Being cyclic means that the head and tail pointers loop back to the beginning once they reach the CRQ's capacity R. This allows for continuous enqueue operations as long as there are available slots. If a CRQ becomes full it gets *closed* from further enqueues, and instead another CRQ

is appended to the LCRQ where the enqueues continue. Dequeue operations are still allowed in a closed CRQ until it is empty and eventually, its memory reclaimed. To locate the actual head and tail the LCRQ itself has a head and tail CRQ where these are located respectively. Inside a CRQ, the enqueue operation involves an increment of the tail position through F&A. Similarly the dequeue operation increments the head position. Utilizing F&A in this way enables multiple threads to be spread out around the slots of a CRQ, concurrently performing CAS on their slot. Although threads may end up contending by performing CAS on the same slot, this is not the common case. Instead, threads move on before trying a CAS by quickly executing a new F&A to retry in a subsequent slot. Figure 2.3 visualizes an LCRQ where the head CRQ is at full capacity and thus closed.

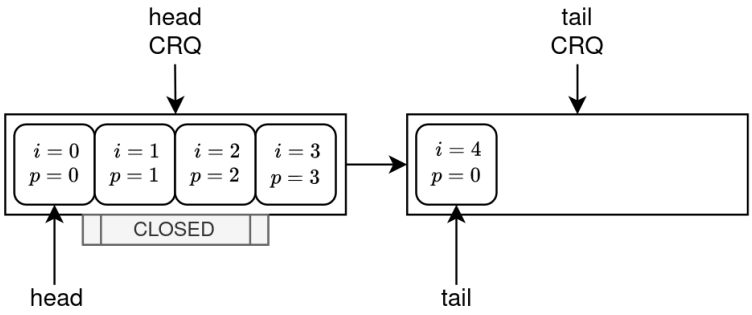

Figure 2.3: LCRQ with $R = 4$. Actual head and tail are located in different CRQs.

Furthermore, because of the cyclic structure of a CRQ, each node, in addition to the actual *value*, is assigned a unique index i. The index is assigned to a node when it is enqueued and is at that moment effectively the tail index, the tail is continuously incrementing for each successful enqueue thus setting a unique index for each node. The position of a node within the CRQ is determined by $p = i \ \% \ R$. While i is unique for each node, p may repeat due to the cyclic nature of the CRQ. This uniqueness of i is crucial for validating enqueue and dequeue operations, as it precisely identifies nodes.

To maintain consistency and order, the algorithm is designed so that a node enqueued by a specific operation enq_i can only be removed by the corresponding dequeue operation deq_i. Through this condition, any enqueue or dequeue of other nodes with the same position p in the CRQ, obtained through $i + kR,\ k > 0$, will fail at position p and retry in the next slot, given that position a node occupies p with index i. To the curious reader, we point to the paper for more details. The paper does not explicitly define the notion of *epochs*, but one can see that i/R could entail which epoch the LCRQ is in, as we will see in the algorithm described in Section 2.2.3. Moreover, as described in Section 2.1.3, the CAS operation is inherently limited to single memory locations. Consequently, because nodes within a CRQ require two memory locations (index and value), CAS2 is utilized for the CRQ algorithm instead of CAS when swapping nodes inside it.

2.2.3 LPRQ

Romanov and Koval challenge the requirement of CAS2 in implementing the state-of-the-art LCRQ concurrent queue algorithm. In their study, they present the LPRQ, a *portable* modification of LCRQ that does not rely on CAS2 [38]. Its portability is achieved by utilizing CAS instead of CAS2, thereby ensuring broader compatibility. Their result shows that the LPRQ provides the same performance as the LCRQ. Notably, their modifications are confined to the CRQ algorithms since it is where the CAS2 is utilized, leaving the LCRQ algorithm unchanged

Furthermore, due to the changes in the algorithm, a portable ring Queue (PRQ) is their modified version of a CRQ. Each node called a *cell* is associated with an *epoch*. Similar to the CRQ algorithm, where epochs are not explicitly mentioned, the PRQ only allows a dequeue operation to remove nodes that were enqueued in the same epoch. Although a cell still requires two memory locations, the algorithm simulates CAS2 with a three-step procedure. Firstly a CAS reserves a cell by swapping the cell value with a thread-unique token, effectively preventing any other thread from succeeding in a corresponding CAS. Now a thread can be said to have locked a cell, is in a critical section, and can consistently perform a CAS on the second memory location, the epoch. The third step is to swap the token to the enqueued value, thus leaving the critical section, and allowing other threads to operate on the cell. We refer to the original paper for a more detailed explanation of how the algorithm works.

2.2.4 Memory Reclamation Techniques

One effective solution to handle memory reclamation for concurrent lock-free queues is through *hazard pointers* [28]. This method is specifically designed to safely handle memory reclamation for lock-free dynamically sized structures. It does so by assigning a unique hazard pointer to each thread that needs to interact with a shared resource. Before a thread accesses the resource, it associates its hazard pointer with that resource, signaling to other threads that it is not safe to reclaim its memory. Once a thread is done with the resource, it clears its hazard pointer. If a thread wants to remove some shared resource it marks the resource as *retired* and saves it to a private list. After a thread has accumulated some number R of retired resources, it will scan the hazard pointers of all other threads for matching memory addresses, if no match is found the memory is reclaimed. In a concurrent queue, when a node is dequeued it is also marked as retired, and its memory will be reclaimed if no thread has its hazard pointer associated with it when the responsible thread performs its scan.

Another technique to solve memory reclamation for concurrent lock-free queues is *epoch-based* memory reclamation [12]. This approach associates a *global* epoch with shared resources which will be *observed* by threads accessing the resource. Once all threads have observed the current epoch it will be incremented. A thread that performs some operation that requires memory reclamation will add the associated object to a garbage list together with the current epoch, marking it as unavailable. Importantly, memory cannot immediately be reclaimed in the current epoch or the

next, memory can only be safely reclaimed after two epochs have passed. The reason for this is that some thread A could have observed the current epoch e before another thread B marked an object O as garbage in e which A holds a reference to. Thread A will therefore still hold a reference to O in $e + 1$ and it is not until A, together with all other threads, have observed $e + 1$, that it is impossible for any thread to hold a reference to O.

Note that the two memory reclamation techniques for concurrent systems presented in this section remove the possibility of the ABA problem occurring since a node can never be reclaimed while threads hold references to it.

2.2.5 Alternative Algorithms

As concurrent lock-free FIFO queues are fundamental data structures in numerous applications, there exist several more algorithms for this type of queue than the two utilized in this project. One example is a queue by Tsigas and Zhang [50] that also utilizes CAS for synchronization, but is implemented as a single circular array instead. Another queue by Ladan-Mozes and Shavit [23], improves the MS queue by replacing CAS with a simple store, creating their novel *optimistic* approach. The Basket Queue by Hoffman, Shalev, and Shavit [16] is also based on the MS queue where each node is instead an unordered group of values called *baskets*. Although not lock-free, Hendler et al. [14] introduce the performative synchronization paradigm *flat combining* which uses a global lock that allows for a single thread to execute several requests on contended hot spots. There are many more, but we believe these are some noteworthy examples.

2.3 Rust Programming Language

For low-level software C and C++ are still the dominant languages and have recently undergone serious modernization. Nonetheless, the responsibility of ensuring memory safety is still placed on the developer, increasing the risk of security vulnerabilities through memory safety violations [19]. The Rust programming language, a relatively new language in the context of systems programming, aims to provide strong static guarantees like memory and thread safety associated with high-level languages, while also maintaining the benefits of low-level languages, such as fine-grained control over data layout and memory management [19, 20]. Moreover, research shows that Rust can compete with C and C++ in terms of energy, time, and memory usage [32]. It is a language that is steadily increasing in popularity and is used internally by Dropbox, Facebook, Amazon, and Mozilla's Firefox [21, 45].

2.3.1 Core Development Environment

Among the tools in the Rust toolchain are the fundamental rustup, rustc, and cargo [43]. Similar to Ruby's rbenv, Python's pyenv, or Node's nvm, rustup is the tool that manages and installs Rust with support for a great number of platforms. Furthermore, rustc is the Rust compiler, and cargo is the package manager. A Rust

project, also called a package, contains a `cargo.toml` file containing the package's dependencies specified by the developer. Cargo downloads and compiles these for your package, and can also upload your package to the Rust community's package registry `crates.io`, where it also retrieves the dependency packages. This approach makes for a clean and simple way to handle dependencies and publish your packages.

2.3.2 Ownership

The way Rust manages memory is through its most unique feature *ownership*. It allows Rust to ensure memory safety without using a GC. The ownership model consists of three rules: (1) each value in Rust has an *owner*, (2) there can only be one owner at a time, and (3) when the owner goes out of scope, the value will be *dropped*. These rules are enforced at compile time by the Rust compiler, thereby not affecting the program's runtime. Even though values are always tied to an owner, it does not always have to be the same owner. The first and second rule implies that values can be transferred to different owners, called a *move*. When a value is moved it becomes inaccessible to the previous owner, unless ownership is moved back to it. Listings 2.1 demonstrates a move where the value is no longer accessible to the original owner.

```
fn main() {
    let original_owner = String::from("hello");
    let second_owner = original_owner;

    // Error! Ownership has been moved to second_owner
    println!("{}, world!", original_owner);
}
```

Listing 2.1: Demonstration of Rust's ownership feature.

Furthermore, when not relying on a GC for memory reclamation, the difficult problem of identifying when memory is no longer being used has to be addressed. This is where the third rule plays an important part, when a value is dropped its memory is also reclaimed [22]. This approach to memory reclamation is deterministic since this rule is enforced at compile time and can also be called *deterministic destruction*. An example of this is in Listings 2.2, the declared variable will only live within the scope it was declared.

```
fn main() {

    {
    // Start of scope
        let var = String::from("hello");
    // End of scope and memory is reclaimed
    }

    // ERROR! var has been dropped and is no longer available
    println!("{}, world!", var);
}
```

Listing 2.2: Example of Rust's deterministic destruction.

2.3.3 References and Borrowing

Always having to move a value can become inconvenient, especially if it is only necessary to read the value and not modify it. To address this, Rust provides *references*. By only granting read access, called *immutable* access, several references to a value can exist. Similar to pointers, references provide a way to access data, even though it is owned by another variable. It refers to a value without taking ownership of it. The difference from a pointer is that a reference is guaranteed to point to a valid value of a particular type for the duration of the reference's scope. The action of creating references is called *borrowing*. Listing 2.3 shows a working version of the example in Listings 2.1 by using a reference instead.

```
fn main() {
    let original_owner = String::from("hello");

    // Reference instead with the "&" operand
    let second_owner = &original_owner;

    // Will print "hello world!"
    println!("{}, world!", original_owner);
}
```

Listing 2.3: Example of creating a reference instead of moving ownership.

However, sometimes it is necessary to also modify a borrowed value and Rust allows this through *mutable references* [22]. Listings 2.4 shows how Rust allows the creation of a mutable reference without transferring ownership.

```
fn main() {

    // We must declare that it will be mutated with "mut"
    let mut original_owner = String::from("hello");

    // Mutable reference instead with the "&mut" operand
    let second_owner = &mut original_owner;

    // Mutable access is allowed to modify
    second_owner.push_str(" world");

    // Will print "hello world!"
    println!("{}!", original_owner);

}
```

Listing 2.4: Mutable reference to modify without taking ownership.

Furthermore, every reference in Rust has a *lifetime* tied to it, which is the scope for which that reference remains valid [22]. These lifetimes are often implicit and inferred, Listing 2.5 illustrates how the Rust compiler implicitly infers lifetimes. The code will however not compile since the lifetime of `x` is smaller than the reference `r`.

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
```

```
5           let x = 5;          // -+-- 'b   |
6           r = &x;             //  |        |
7       }                       // -+        |
8                               //           |
9       println!("r: {}", r);   //           |
10  }                           // ----------+
```

Listing 2.5: Lifetime of two variables. Lifetime 'a and 'b are respectively tied to r and x.

2.3.4 Type System

Every value is of a certain data type and Rust requires that the type of every value is known at compile time. This condition, combined with the principles of ownership and borrowing, forms Rust's stringent type system. In more formal terms: a value of a certain type can either be accessed by multiple immutable references or modified through a unique mutable reference. This restriction is uncommon in other programming languages, where variables can be mutated arbitrarily, and is how Rust prevents data races at compile time [22, 53].

2.3.5 Memory Allocation

The memory that is available during runtime for Rust programs is divided into the *heap* and *stack* [22]. Values placed on the stack are ordered with the *last-in-first-out* (LIFO) principle and these values must therefore be of a known fixed size. Typical values that are placed on the stack are integers and pointers. The heap on the other hand is less organized because it holds all dynamically sized values.

2.3.6 Concurrency in Rust

One of Rust's major goals is to safely and efficiently handle concurrent programming. Through its ownership model and type system, Rust is capable of identifying many concurrency-related errors at compile-time. Consequently, this means developers can save time during development since they do not have to reproduce exact runtime scenarios trying to catch these errors [22].

Furthermore, naturally, for the queues in this paper, threads will simultaneously perform enqueue and dequeue operations. Being lock-free queues necessitates multiple mutable references, which violates the fundamental rule described in Section 2.3.4. Consequently, this is one reason unsafe code is required to implement such a data structure, and it is up to the developer to secure it [3]. Additionally, as mentioned in Section 2.3.3, references cannot be null. However, in a linked list, it is essential for a node to potentially reference the next node that may not exist, a null pointer, indicating the end of the list. This requirement is another reason why unsafe code is needed. Although unavoidable, unsafe code should only be applied where it is strictly necessary, preserving the safety guarantees for the rest of the program.

Rust and its community provide libraries to ease the development of concurrent applications. Rust's standard library includes the synchronization library `sync` [44]

which provides a variety of tools designed to handle concurrency. Among these are the `atomic` module and the `Arc` type. As the name suggests, `atomic` provides primitives such as atomic pointers, integers, and booleans, each equipped with methods ensuring atomicity, including CAS and F&A. The `Arc` type wraps another type and utilizes atomic operations to safely share ownership between threads to the wrapped type. Importantly, `Arc` does not provide atomic access to the wrapped type, it only allows shared ownership between threads. It does so through *atomic reference counting*, which means that a separate counter is atomically incremented for each created reference to the wrapped type's memory. `Arc` wrapped types are only allowed to be dropped when the last reference to it is dropped, i.e., the separate atomic counter is zero. Although a simple solution to ensure safe memory management in a concurrent system, reference counting is slow, and will struggle under the contention arising from high levels of concurrency [22, 39, 49]. Another useful library, though not part of the standard, is Crossbeam [8] which provides additional tools for concurrent applications.

2.3.7 Miri

Miri (Mid-level Intermediate Representation Interpreter), is a specialized Rust interpreter developed to aid Rust programmers in identifying and mitigating undefined behavior (UB) arising from unsafe code [18]. By executing Rust code in a controlled environment, Miri analyzes the runtime behavior of the program. This allows Miri to identify incorrect program behavior originating from unsafe code blocks. Moreover, among the types of UB that Miri is capable of identifying, those relevant to a concurrent queue include unsafe memory access, and, although still in an experimental phase, data races.

Even though complex in its inner workings, installing and using Miri is simple. It is easily installed through a single rustup command which will handle the download and your Rust code can thereafter be executed under Miri by just including the `miri` keyword. For specific test code, the `cargo miri test` is used, and for binary projects `cargo miri run`.

2.3.8 Strict Pointer Provenance

The Rust language is under active development and some useful features are not yet stable and are therefore only available in unstable releases, known as *nightly releases* [46]. Among these features is the strict pointer provenance, an experimental framework to ensure safety for pointers in Rust [47]. Without requirements on provenance, it is possible to convert any integer to a pointer type and use it as a pointer. On the other hand, with strict pointer provenance, to access some location through a pointer that pointer is required to have an unbroken chain of custody back to its original allocation, ensuring the chain is compromised solely of pointer-to-pointer operations.

Since this feature is available only on nightly releases of Rust, the project needs to explicitly specify the use of the nightly toolchain. A convenient way to do this is to

create or modify the `rust-toolchain.toml` file in the project's root directory. Once the nightly release is specified, the strict pointer provenance feature can be enabled by including `#![feature(strict_provenance)]` in the code, usually in the main source file.

2.3.9 Data Struct Layout

Programming languages usually store the fields of structs in adjacent locations in memory [39]. In Rust, the layout for user-defined structs is specified by a *representation*. Importantly, in the default representation, the actual memory layout does not need to match the order in which the fields are declared in the code. However, Rust provides the capability to override the default by explicitly specifying a representation that is compatible with the C language. This representation can be leveraged to achieve fine-grained control over the data layout [48].

The ability to exactly control how data is laid out in memory allows for cache access optimization [10]. For synchronization reasons, updates on atomic values invalidate the whole line in the cache it belongs to [9]. For the queues in this paper this means that if both head and tail are on the same cache line, both will be invalidated when updating either one through a successful CAS or F&A. Consequently, the program must spend extra time synchronizing. However, this can be avoided by utilizing the available control over layout, ensuring that head and tail fields are padded to the length of a cache line which automatically places them on separate lines in the cache.

2.3.10 Memory Model

Depending on the hardware and operating system, the way memory is managed can differ, each utilizing different *memory models* [2]. Understanding how these models impact concurrent programs is important, though they can be quite complex and hard to understand in modern systems [26]. Therefore, programming languages define their own, simpler memory model to align with the range of models the language aims to support [49]. These simplified models allow developers to better understand and manage program execution, particularly concerning compiler optimizations like re-ordering.

Rust is no exception to this and its memory model is directly based on the C++11 model, which leads to many similarities between them [49]. Rust's memory model allows programmers to explicitly specify the order in which atomic operations should be executed, called *memory orderings*. In other words, these orderings specify to the compiler the order in which the concurrent program's atomic operations should be ordered to ensure that all atomic operations, accessing the same shared resource, comply with what the program intends to do. For instance, without any memory order, a compiler re-ordering could change the whole dynamic structure of a concurrent program. One example is the most strict ordering, a *sequential consistent* order. Atomic operations under sequential consistency will always be executed in the sequential order as they are in the code with respect to each other, the compiler

is not allowed to shuffle them around. The pseudocode in Listing 2.6 illustrates the uncertainty of a value without memory ordering. With sequential consistent order for all atomic operations, the value would have been guaranteed to be 2.

```
fn main() {
    let x: 1;
    let y: 2;
    let atomic_var: atomicInt(0);

    atomic_var.store(x, Ordering::None);
    atomic_var.store(y, Ordering::None);

    // result can be 0, 1, or 2 due to re-ordering.
    let result = atomic_var.load(Ordering::None);
}
```

Listing 2.6: Example without memory ordering.

Another way to ensure the compiler does not re-order code is through *memory fences* [15]. This approach essentially forces a sequential consistency throughout its whole encapsulated scope. Although this approach can significantly ease the process of finding synchronization bugs, it is expensive in terms of time.

3

Methods

3.1 Rust implementations

Both the MS Queue [29] and LPRQ [38] implementations were done by carefully analyzing the pseudo-code and reading the algorithm specification in the corresponding paper. Additionally, the external C implementation of MS Queue by Chaoran Yang [52] and the C++ version available in the LPRQ paper's repository [37] served as a valuable source of inspiration, providing a way to clear any ambiguities encountered during the development. The Rust implementations, along with the benchmarking suite and the C and C++ references, are available in our public GitHub repository [17]

3.1.1 Technical Design Choices

Selecting the LPRQ over LCRQ was based on three factors. Primarily, the LPRQ paper shows it is possible to preserve the LCRQ performance by relying on CAS instead of CAS2. Secondly, the improved portability across different platforms. Lastly, while still possible to implement CAS2 in Rust [42] it is not as explicit as CAS, so we regarded it as more implementation-friendly to consistently use CAS over CAS2.

Important design decisions were made to effectively implement these algorithms, such as what libraries to rely on and what optimizations are required. As the algorithms inherently rely on atomic operations the `atomic` module in `sync` was included in the design to obtain appropriate primitives that support CAS and F&A operations. Moreover, due to Rust's ownership rules it was necessary to wrap both queue structures within the `Arc` type to enable multiple threads to concurrently operate on it.

Another design choice was related to cache line optimization, an optimization also adapted by the LPRQ paper. To implement this we utilized the Crossbeam library, which provided an efficient way to fill the queue's head and tail with padding to fit a whole cache line [8].

3.1.2 Memory Management

As for memory management, we utilized hazard pointers for the MS queue since it has shown to be a suitable choice, as mentioned in Section 2.2.1. It is also the same

technique used by the C reference implementation.

For the LPRQ implementation, we included two versions, one with hazard pointers, and the other with epoch-based memory reclamation. Hazard pointers were chosen since the LPRQ paper explicitly mentions it as a solution for manual memory management. Integrating this technique with our implementation was done by leveraging the existing library Haphazard, adapted from Meta's Folly library [13]. The Crossbeam's epoch library was utilized to integrate the epoch-based version. This alternating version allowed us to compare the Haphazard version with another robust technique, which gave us some insight into the performance of the underlying LPRQ algorithm.

Additionally, a third LPRQ version was included to explore how a more idiomatic Rust version would work. This version used the library Aarc which adds a version of Rust's atomic reference counted smart pointers that support CAS operations. While reference counting causes much higher contention and thus usually worse performance, it has the upside of removing the need for manual memory management entirely. The library does not however use the same reference-counting approach as the Rust standard library [51], but instead uses the deferred reference counting method proposed in [1].

3.1.3 Library adaptation

To get more comparable results between Rust and C++ we modified the Haphazard library removing a memory fence it uses after every change to what a hazard pointer is protecting. In general, this fence is necessary for soundness but in this case, it is not needed and the C++ version does not use anything similar. An alternative would have been to modify the C++ reference to add similar fences, but that would have been a much more involved change.

3.2 C/C++ References

For the Michael and Scott queue reference implementation we used Chaoran Yang's `fast-wait-free-queue` C library [52]. This was chosen due to it using hazard pointers for memory reclamation and having a built-in benchmark which was easy to adapt to match our Rust benchmarks. Furthermore, the LPRQ paper's repository [37] served as the reference implementation for this queue, providing an LPRQ implementation in C++. Similar to the `fast-wait-free-queue`, it also utilizes hazard pointers and contains a benchmark suite, which could be adapted to our benchmarks.

3.3 Benchmarks

Inspired by the evaluation methodology in the LPRQ paper [38], a pairwise and Multi-Producer Multi-Consumer (MPMC) workload was used in the benchmarking

process. However, only the pairwise was deemed necessary for the MS Queue since it served primarily as a proof of concept.

In both workloads, several threads operate on a shared concurrent queue simultaneously and the work is divided evenly among the threads. To make it more realistic, each thread could be delayed between operations simulating some other activity. Furthermore, we focused on keeping the benchmark code as consistent as possible between the two languages. This approach allowed us to, with more certainty, attribute any observed differences in performance directly to the implementations, rather than to variations in benchmark design.

We used the command line tool `hyperfine` [33] as our benchmarking tool which measures the average runtime of a given benchmark binary. This average runtime allowed us to calculate the throughput, which is the measurement used in the LPRQ paper. Additionally, to provide a robust assessment of each language's performance, we monitored energy consumption, memory usage, and access patterns. Since `hyperfine` does not include a way to collect information about these additional metrics, the performance analyzing tool `perf` [25] was used for energy consumption and memory access behavior, while memory usage was captured with `memusage` [27].

Importantly, as `hyperfine` measures the total runtime of a binary, all benchmarks start with an empty queue and threads start to work on the queue as soon as they are created, and not synchronized behind a barrier before they start. Including a barrier or performing a pre-fill would just delay the resulting runtime given by `hyperfine`.

Each result from a benchmark was normalized relative to the language that demonstrated the highest efficiency in each specific benchmark, providing an overview of each language's performance.

3.3.1 Hardware

The benchmarks were run on a system based on the x86-64 architecture to ensure support for the relevant atomic operations. The system consists of two sockets each equipped with a 2.10 GHz Intel(R) Xeon(R) CPU E5-2695 processor. Each processor has 18 cores, each of which multiplexes 2 hardware threads resulting in 72 in total on the system. The cache architecture of each core consists of 32KB L1 instruction and data caches, 256Kb L2 cache, and a shared 45MB L3 cache.

Moreover, to avoid unnecessary synchronization overhead between the two processors we pinned the benchmark executions to threads belonging to a single socket, effectively utilizing only 18 cores. Since each core can multiplex two threads, cores will start hyperthreading when the benchmarks utilize over 18 threads, allowing for 36 threads in total. This process was done by capturing the core's identifier and only running the benchmark if it belonged to the same processor. In our case, cores with an even identifier belonged to the same processor.

3.3.2 Workloads

For the pairwise workload, each thread is tasked with performing pairwise enqueue and dequeue operations, sequentially executing an enqueue operation followed by a dequeue. Listings 3.1 shows a simplified pseudocode for the pairwise benchmark.

```
fn pw_benchmark(threads, operations, queue) {

    for threads {
        thread::spawn(
            for operations {
                queue.enqueue();
                delay(100 cycles);
                queue.dequeue();
                delay(100 cycles);
            }
        );
    }
    thread::join_all();
}
```

Listing 3.1: Pairwise Pseudocode.

In the MPMC workload, several producer threads enqueue elements to a shared queue, while some consumer threads continuously try to dequeue. Once the producers are finished the consumers are notified through an atomic boolean variable, and they eventually stop when the queue is empty. A simplified version in pseudocode can be seen in Listings 3.2. This workload was only applied to the LPRQ.

Due to relying on `hyperfine` to measure the total runtime, we could not wait until all threads were created, so we decided to create the producer threads before the consumer threads, avoiding as many unnecessary empty dequeues as possible.

```
fn mpmc_benchmark(producers, consumers, operations, queue) {

    for producers {
        thread::spawn(
            for operations {
                queue.enqueue();
                delay(100 cycles);
            }
        );
    }

    for consumers {
        thread::spawn(
            loop {
                if queue.dequeue().isNone() {
                    if stop_consuming {
                        break;
                    }
                }
                delay(100 cycles);
            }
        );
    }
```

```
24      thread::join_all(producer_threads);
25      stop_consuming.store(true);
26      thread::join_all(consumer_threads);
27  }
```

Listing 3.2: Multi-producer multi-consumer Pseudocode.

The two workloads were compiled into separate benchmarking binaries for each implementation, including configurable parameters. Specifically, the pairwise binary allowed adjustments in the number of threads, and operations. For the MPMC binary, the number of producer and consumer threads could be adjusted.

3.3.3 Benchmark configurations

The configurations for the pairwise workload were created by raising the number of threads from one to the maximum amount of threads but with a fixed number of operations at 10^8. Testing with different amounts of threads allowed us to see each language's throughput with an increasing concurrency level.

Furthermore, we used two producer-consumer ratios, 1:1 and 2:1 creating two benchmarks for the MPMC workload. Using two ratios allowed us to analyze the impact different balances had on the throughput. Similar to pairwise, these benchmarks used 10^8 operations for each configuration and also increased the threads for each configuration while maintaining the ratio between producer and consumer threads. However, importantly, the 10^8 operations for this workload are the enqueue operations performed by the producers, since consumers will continuously dequeue until signaled to stop, see pseudocode in Section 3.3.2.

The 1:1 ratio can be seen as a baseline for how the system performs under an equal load distribution. The latter could indicate how each language would perform in an environment where data is gathered faster than it can be processed, such as a logging system.

Similarly, the energy consumption and memory benchmarks with `perf` and `memusage` utilized the max amount of threads and 10^8 operations. These benchmarks were only executed with the MPMC workload's two ratios as it resembles a real-world scenario more than the pairwise workload.

To ensure statistical significance, each configuration was executed 10 times for the `hyperfine` and `perf` benchmarks, while `memusage` was executed once since it does not include a parameter to adjust the number of executions.

4

Results

This chapter outlines the benchmarking results, providing insights into each language's performance using our implementation. In the graphs, the y-axis displays the throughput of each benchmark, while the x-axis shows the number of threads. Each point in the graphs includes the standard deviation. The values in the tables are normalized relative to the most efficient language, the top value corresponds to the most efficient language's actual value in each table. The raw values from the MPMC benchmarks are found in Appendix A, and the raw results from all the benchmarks are available in our public GitHub repository [17].

4.1 MS queue

Figure 4.1 shows the results from the MS queue benchmarks under the pairwise workload with increasing threads. The Rust implementation is consistently slower than the C reference with an increasing concurrency level. This might be attributed to variations in hazard pointer implementations or the safety guarantees provided by Rust. Additionally, the MS queue declines in throughput as the number of threads increases, as expected.

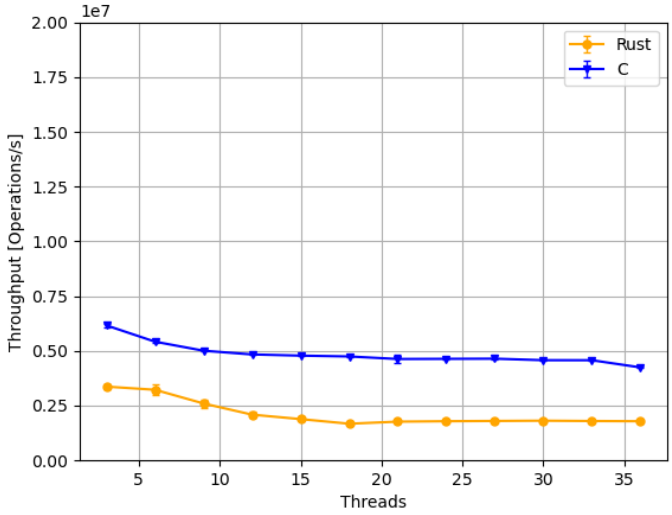

Figure 4.1: MS queue under pairwise workload with increasing concurrency, each step performing 10^8 operations.

4.2 LPRQ

This section presents the benchmarking results of the LPRQ implementation. First, we present the pairwise workload. Secondly, we present the MPMC benchmarks with the 2:1 ratio being the first of the two as we consider it to be the closest to a real-world scenario. Additionally, the epoch version is not included for any of the 2:1 benchmarks for reasons mentioned in Section 5.2.

4.2.1 Pairwise

Under the pairwise workload, Rust versions without `aarc` initially outperform the C++ version, as seen in Figure 4.2, but at higher levels of concurrency, the C++ version has slightly better throughput. Although the variance is prominent for all versions except `aarc`, the increase in throughput is what could be expected from the LPRQ algorithm. Additionally, the trade-off of using the idiomatic Rust version with `aarc` is apparent.

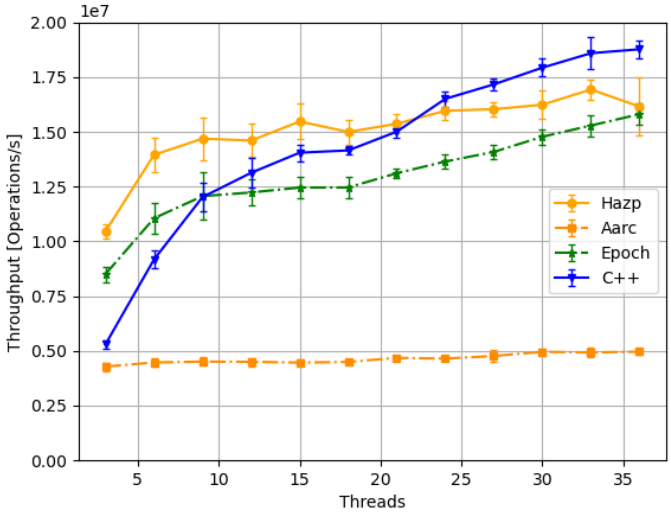

Figure 4.2: LPRQ under pairwise workload with increasing concurrency, each step performing 10^8 operations.

4.2.2 MPMC 2:1 ratio

The results from the MPMC throughput benchmark with the 2:1 ratio can be seen in Figure 4.3. The C++ version is the most consistent with what could be expected from the LPRQ algorithm as the concurrency level increases. Notably, at the highest concurrency level, the Rust implementation with hazard pointers eventually achieves comparable throughput. Again, the sacrifice of throughput for the benefits associated with the `aarc` version is clear.

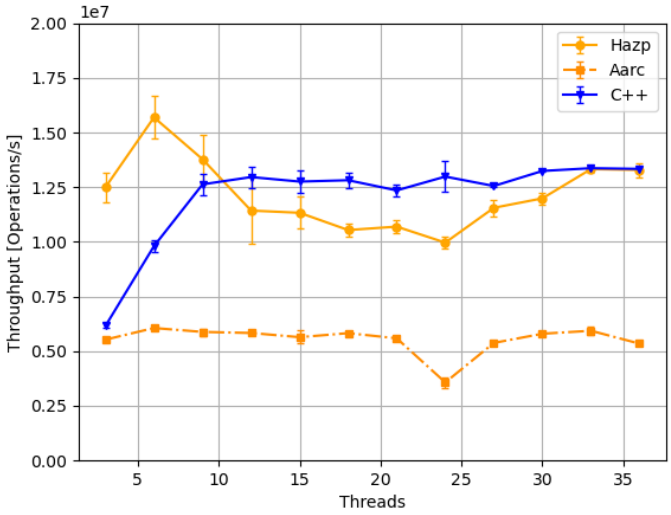

Figure 4.3: LPRQ under 2:1 MPMC workload with increasing concurrency, each step performing 10^8 operations.

Table 4.1 gives an overview of the time and energy benchmarks with the normalized values. Rust and C++ utilizing hazard pointers are almost identical for these two measurements. Again, the overhead from utilizing `aarc` is expected, resulting in additional time and energy. In Table 4.2 we can see the Rust versions have a significantly larger peak stack usage than C++, while the peak heap usage remains fairly similar. These two benchmarks were each executed with 36 threads, the highest level of concurrency, with 10^8 operations. The time and energy results are the average from 10 benchmark executions, while the memory usage benchmark was executed only once.

Table 4.1: Time and energy for 2:1 MPMC benchmark.

Time	
7.5 s	
C++	1.0
Hazp	1.0
Aarc	2.49

(a) Runtime.

Energy	
1056 J	
Hazp	1.0
C++	1.03
Aarc	2.67

(b) Energy consumption.

Table 4.2: Memory usage for 2:1 MPMC benchmark.

Heap peak	
67237 B	
Hazp	1.0
C++	1.37
Aarc	1.67

(a) Peak heap usage.

Stack peak	
10272 B	
C++	1.0
Hazp	25.69
Aarc	26.45

(b) Peak stack usage

Furthermore, the different versions' memory access behavior can be seen in Table 4.3. These were also executed with 36 threads with 10^8 operations, and the results are the average of 10 benchmark executions. Although both Rust and C++ with hazard pointers show a similar number of cache references, C++ experiences significantly more cache misses, indicating the Rust implementation manages cache access more efficiently. Comparably, the `aarc` version likewise suffers from many cache misses, however, it also makes more references to the cache. Additionally, unlike the other two versions, `aarc` also generates a huge amount of page faults, perhaps indicating that the extra atomic operations associated with reference counting are the cause. This considerable difference between the two Rust versions underscores the suitability of hazard pointers for concurrent FIFO queues, mentioned in 2.2.4.

Table 4.3: Memory access behavior for 2:1 MPMC benchmark.

Page fault	
75529	
Hazp	1.0
C++	4.86
Aarc	33.07

(a) Page fault.

Cache reference	
2.38E9	
Hazp	1.0
C++	1.51
Aarc	8.86

(b) Cache reference.

Cache miss	
1.16E6	
Hazp	1.0
C++	20.03
Aarc	22.72

(c) Cache miss.

4.2.3 MPMC 1:1 ratio

Figure 4.4 shows the throughput benchmarking result from the MPMC workload with a 1:1 ratio. Again, the C++ version is what is somewhat expected from the LPRQ's scalability. The Rust versions all peak early and then decline in throughput, indicating an intrinsic performance bottleneck not present in the C++ version. Though worst in performance, the `aarc` version does not fluctuate as much as the other two Rust versions.

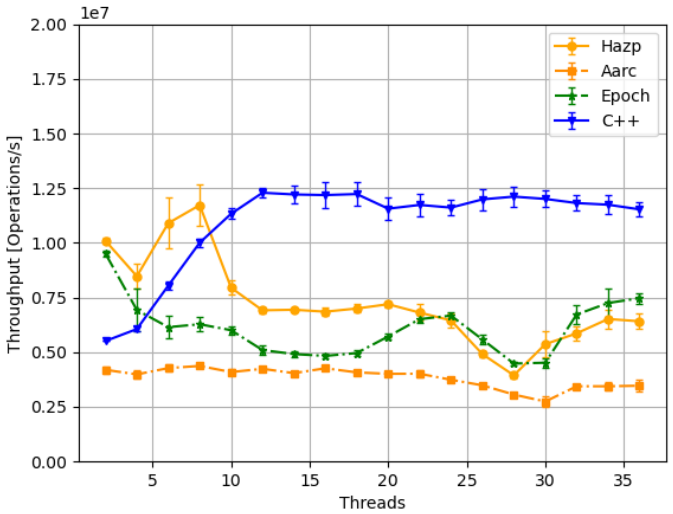

Figure 4.4: LPRQ under 1:1 MPMC workload with increasing concurrency, each step performing 10^8 operations.

Similar to the 2:1 ratio benchmarks, time, energy, and memory benchmarks for the 1:1 ratio were executed at the highest concurrency level with 10^8 operations, and the same number of times for the respective benchmarks. An overview of time and energy can be seen in Table 4.4, memory usage in Table 4.5, and memory access pattern in Table 4.6. In this workload and ratio, it is clear that C++ is outperforming the Rust versions.

Table 4.4: Time and energy for 1:1 MPMC benchmark.

Time	
8.68 s	
C++	1.0
Epoch	1.54
Hazp	1.8
Aarc	3.34

(a) Runtime.

Energy	
1259 J	
C++	1.0
Epoch	1.58
Hazp	1.81
Aarc	3.16

(b) Energy consumption.

Table 4.5: Memory usage for 1:1 MPMC benchmark.

Heap peak	
46597 B	
Hazp	1.0
Epoch	1.72
Aarc	1.81
C++	1.98

(a) Peak heap usage.

Stack peak	
10272 B	
C++	1.0
Hazp	25.69
Aarc	26.45
Epoch	26.81

(b) Peak stack usage.

Table 4.6: Memory access behavior for 1:1 MPMC benchmark.

Page fault	
1321	
C++	1.0
Aarc	9.0
Hazp	14.92
Epoch	17.42

(a) Page fault.

Cache reference	
2.8E9	
Epoch	1.0
C++	1.37
Hazp	1.57
Aarc	2.56

(b) Cache reference.

Cache miss	
2.4E5	
C++	1.0
Hazp	2.2
Epoch	2.48
Aarc	2.59

(c) Cache miss.

5

Ergonomics

This section presents our experience using Rust in this project, with a focus on developer ergonomics compared to lock-free programming in C++.

5.1 Standard library support

Although slightly different in their interface designs, the atomic primitives and operations are very similar between the Rust and C++ standard libraries. Relevant to this is that their memory models are almost identical as well, this is due to Rust's ordering semantics being defined as following the C++20 standard.

One of these design differences that is more beneficial in C++, is that the C++ `std::atomic<>` can be used with custom types. As long the type meets certain requirements the compiler will automatically figure it out. Rust, on the other hand, only provides atomic versions of primitive types, with no built-in support for user-defined types without writing wrapper types around these primitives. This can increase the risk of introducing bugs, as programmers have to manually ensure the safety of the created wrappers. Luckily, this did not make a difference in this project since the queues only used primitive atomic pointers and atomic integer types. However, we believe it is still noteworthy as it makes it easier to create custom types with atomicity in C++ than in Rust. Even though less flexible, most operations for these primitives in Rust have more or less the same interface and in many cases even the same names.

As mentioned in 2.2.3, the LPRQ does not make use of the double-width CAS2 as the LCRQ algorithm. This instruction is the core reason for choosing LPRQ over LCRQ because neither Rust's nor C++'s standard library offers a simple way to use it. However, as C++ is more flexible, it is easier to use it.

In C++, GCC and Clang use `libatomic` to handle the actual implementations of atomic operations, and it supports 128-bit CAS, namely `std::atomic<__int128>`. Therefore, implementing the CAS2 can be done by utilizing this primitive, or simply by placing two 64-bit values in a struct, effectively delegating it to the compiler. On x86-64 architectures this should compile to a single CAS2 instruction, but the compiler is allowed to implement it using mutexes if it wants to. Consequently, to guarantee that it is lock-free, C++ provides the `std::atomic<T>::is_lock_free()` which has to be used to check at runtime. Because of this, it is still fairly common

to use inline assembly for CAS2, such as in the LPRQ papers LCRQ reference, effectively minimizing the difference between Rust and C++.

For Rust, the low-level `core::arch` contains CPU and vendor-specific functions, including the CAS2 function `core::arch::x86_64::cmpxchg16b`. However, this is mostly a thin wrapper around the assembly instruction, meaning it is not supported by any built-in 128-bit atomic primitive. Using it would therefore require the programmer to manually ensure the 128-bit primitive is safe to use, which can be quite an involved process. Consequently, it requires an unsafe block, which the CAS for the supported, built-in, atomic primitives does not.

Furthermore, one thing the Rust standard library does better than C++ is the use of sum types which increases code readability. This type can take on a single variant out of different available variants. A relevant example is the Rust `compare_exchange()` function which returns the sum type `Result<T, T>`. This type has two variants, `Ok(val)` and `Err(val)`, which indicate if the operation succeeded or not. The value contained in the `Result<T, T>` is either the supplied value in case of success or the actual current value in case of failure. Since the C++ standard library does not use sum types it has to return the current value as an out parameter. However, this is generally discouraged since it leads to less readable code, especially since C++ does not have a way to indicate at the call site if an argument is modified or not, compared to mutable references in Rust which are used when values are most likely going to be modified.

In Listing 5.1 is an example of how Rust's `Result` type is used in the `enqueue` method of the LPRQ, we use this to try to help advance the tail pointer after a failed attempt to add a new PRQ to the LPRQ. In the C++ reference, a reference needs to be initialized separately, where the CAS will place the relevant value, the out value. The C++ reference chooses to not use this out parameter and instead loads `next` separately. Notably, the Rust version matches the pseudo-code from the paper closer than the C++ implementation, suggesting that this is a more intuitive way of doing it. In this specific example `queue.next` is a Haphazard atomic pointer, not a standard library one, but the return type is the same.

```
match unsafe {
    queue.next.compare_exchange_ptr(
        ptr::null_mut(),
        new_tail_ptr
    ) } {
        Ok(_) => {
            ...
        }
        Err(next) => {
            let _ = unsafe {
                self.tail.compare_exchange_ptr(
                    queue_ptr.cast_mut(),
                    next
                )
            };
            // Drop the failed new tail so it does not leak
            let _ = unsafe { new_tail.retire() };
            continue;
```

```
19          }
20  }
```

Listing 5.1: Compare-exchange example from enqueue

In summary, Rust's standard library does provide what is needed for lock-free programming. However, it might comparably be less convenient since C++ generally provides more flexible atomic types and a higher-level interface for CAS2. However, Rust's standard library tends to be more explicit. Most atomic operations require the caller to specify the desired memory ordering, compared to C++, which automatically sets sequentially consistent as the default. One could argue that being explicit in this area could assist developers in understanding additional underlying complexity, and room for optimization.

5.2 Memory reclamation

As already mentioned, both Rust and C++ lack automatic GC, and therefore some manner of concurrent memory reclamation needs to be implemented manually for the queues. Our experience using the three Rust libraries, Haphazard, Crossbeam's epoch, and Aarc, offered valuable insights into the language and the ease or difficulty of integrating an existing library.

Haphazard is a fairly standard hazard pointer implementation, but it does use some Rust features to try and add some extra safety. One example is that the references returned from a load are tied to the hazard pointer's lifetime, which prevents a reference from outliving it. For example, the code in listing 5.2 does not compile since `hazard` does not live as long as the reference `a_ref`.

```
1   let a = AtomicPtr::from(Box::new(13));
2   let mut a_ref: &i32;
3   {
4       let mut hazard = HazardPointer::new();
5       a_ref = a.safe_load(&mut hazard).unwrap();
6   }
7
8   // ERROR! a_ref was tied to hazard's lifetime
9   println!("{}", a_ref);
```

Listing 5.2: Erroneous code showing that a loaded reference can not outlive its hazard pointer.

Additionally, since the `safe_load()` method takes a mutable reference, Rust's borrowing rules prevent changing what the hazard pointer is protecting while still holding the reference. Listing 5.3 demonstrates how Rust's borrowing rules protect the programmer from, by mistake, bypassing a fundamental safety rule by simply using Haphazard.

```
1   let a = AtomicPtr::from(Box::new(13));
2   let b = AtomicPtr::from(Box::new(31));
3
4   let mut hazard = HazardPointer::new();
```

```
5
6       // First borrow.
7       let a_ref = a.safe_load(&mut hazard).unwrap();
8
9       // Second borrow.
10      let b_ref = b.safe_load(&mut hazard).unwrap();
11      println!("{}", b_ref);
12
13      // ERROR! Only one mutable reference is allowed.
14      println!("{}", a_ref);
```

Listing 5.3: Incorrect reborrow.

A correct version that complies with the borrowing rules can be seen in Listing 5.4. The compiler will notice that the first mutable reference goes out of scope and releases the borrow, which allows the second borrow to successfully load and create a mutable reference to the protected value.

```
1       let a = AtomicPtr::from(Box::new(13));
2       let b = AtomicPtr::from(Box::new(31));
3
4       let mut hazard = HazardPointer::new();
5
6       // First borrow
7       let a_ref = a.safe_load(&mut hazard).unwrap();
8       println!("{}", a_ref);
9
10      // Second borrow
11      let b_ref = b.safe_load(&mut hazard).unwrap();
12      println!("{}", b_ref);
```

Listing 5.4: Correct reborrow.

All of this is checked at compile time, so there are no runtime costs for this extra safety. We believe this is a good example of how Rust features can be leveraged to make safer interfaces.

Using Haphazard was straightforward, and despite it needing quite a few unsafe functions, the documentation was very clear about what the safety preconditions were, with the most common one just being not manually freeing anything the library was managing, or making sure pointers were aligned. We only had a few bugs related to memory reclamation with Haphazard, and they were easily found by running under Miri. One trade-off the Folly style of hazard pointers has is that it is not entirely predictable when a reclamation happens. When an object is retired the library checks if enough time has passed since the last reclamation and in that case does a pass. However, this is still more predictable than a traditional GC since the reclamation pass can only happen right after a call to the retire function, but it is still slightly harder to reason about than normal deterministic destruction.

Crossbeam's epoch library had good performance, despite these types of queues traditionally lending themselves more to hazard pointers. This library however caused us by far the most issues. Debugging concurrent memory reclamation of any kind is challenging since a thread causing a segmentation fault is rarely the thread

where the bug is. The structure of Crossbeam's epoch made it more difficult than the other versions' libraries to find the root causes of memory bugs. Furthermore, it is the only library we used that could not run under Miri without disabling most of its features, meaning our tests would pass under Miri but still cause a segmentation fault at runtime. We did not manage to resolve one of these bugs despite significant effort, which is why the epoch version is not present in the 2:1 ratio benchmark, as it would sporadically cause segmentation faults when the queue got too full. Since the library is widely used, it is almost certainly a matter of us misusing it in some way, but due to time constraints, we could not investigate it further. Notably, the debugging became much more difficult without Miri's highlights, which emphasize the value Miri provides when writing unsafe code. Without Miri, the debugging experience is almost identical to C++ projects.

While the version based on Aarc is significantly slower than the other two versions, it comes with the benefit of having an extremely easy-to-use API, in contrast to our experience using Crossbeam's epoch. Importantly, it has no public unsafe functions. Hence, it is impossible to violate any of Rust's safety guarantees by misusing the library, whereas both Haphazard and Crossbeam's epoch have unsafe functions needed for basic use. While the performance trade-off is probably not worth it in production code, Aarc might be useful for prototyping lock-free algorithms since you don't have to worry about incorrect memory management. Since it is safe, easy to use, and idiomatic Rust the library's functionality could reasonably be integrated into the standard library.

5.3 Profiling and Debugging tooling

Except for Miri, the tooling we used for profiling and debugging is similar to what you would use for a similar C++ project. As these tools worked for Rust as well, being an experienced C++ programmer is quite useful when starting Rust compared to other languages. These tools include gdb, Valgrind, and address sanitizer, all tools commonly used in C++ development. Rustup even includes a convenience script `rust-gdb` which takes care of finding the correct debugging symbols and sets some default settings to make debugging Rust code easier. We believe this shared tooling is a huge benefit since you can leverage previous experience even if a team is new to Rust.

For profiling our code, apart from the large-scale benchmarks we developed, we also utilize `perf` to get more granular insight into what parts of our code were taking time. It was helpful to be able to use the same tool to inspect our reference implementations when identifying differences and similarities between the languages. Interestingly to our use case, this is how we found the unnecessary memory barrier in Haphazard, which was causing significant slowdowns.

Though already mentioned, Miri is a tool that has been a vital part of our development process. Its ability to catch a variety of error types arising from unsafe code has been extremely helpful and time-saving. Combined with strict pointer provenance, Miri can even reason well about pointer-to-integer casts and similar error-prone op-

erations. Together they act as a more powerful version of Valgrind's memcheck tool, making it much easier to catch subtle memory issues as well as potential edge cases of undefined behavior. Miri is unfortunately still only available on nightly releases of Rust due to how tightly it integrates with the compiler. However, since Rust maintains strict backward compatibility, a project that sticks to a stable version for its builds can still have a separate nightly toolchain for running Miri.

5.4 Build tools and dependency management

While not specific to lock-free programming, one area where Rust stands head and shoulders above C++ is with its build tools and dependency management. While C++ lacks any unified build system or package registry, the Rust ecosystem is built around cargo and `crates.io`. Compared to CMake, adding dependencies is just a matter of adding a single line to `cargo.toml`, and cargo will download and compile the library. This even works for Rust dependencies not on `crates.io`, as cargo can fetch any git repository, either local or remote as long as it has a `cargo. toml` file in its root.

Similarly, having a built-in unit and integration testing framework saves significant work. Furthermore, Miri integrates well with this test framework, letting you run all the tests under Miri by just changing `cagro test` to `cargo miri test`. Having all of the ecosystem built around one set of tools simplifies most aspects of development.

6

Conclusion

6.1 Discussion

The goal of this thesis was to provide an assessment of Rust's suitability for high-performance concurrent lock-free programming. This included an investigation of how much of Rust's safety guarantees affect performance compared to C and C++ and estimating the extent to which these guarantees must be sacrificed through necessary unsafe code. This section discusses how accurately our chosen methodology and its results can answer these questions.

6.1.1 Benchmarking methodology

When comparing our results to similar ones in the literature, we notice our throughput has a high variance. This could indicate some problems with our measurement methodology. In the LPRQ paper, they measure the throughput over time, whereas we measure the runtime of a specific number of operations divided evenly among threads. This means their throughput will be measured under a continuous flow of operation on the queue from all threads. Our approach, however, could allow one or several threads to perform all their operations quickly before other, slower threads. Consequently, our results could be misleading if there is any significant startup or tear-down time, or if the queue is slower at the beginning or end of a benchmark run.

6.1.2 Safety trade-off

One key question at the start of this project was how much of Rust's additional safety guarantees are still applicable to lock-free programming since it necessarily involves unsafe code. While the guarantees are weaker, they can sometimes be reintroduced as seen in Haphazard's use of lifetimes. During software testing, Miri, especially together with strict pointer provenance, assisted with verifying that direct pointer manipulations did not violate Rust's memory rules. While Miri is not perfect and does not support all Rust features, only the epoch version could not run under Miri, and it caught all memory bugs in the other versions. If the unsafe parts of the program do not contain any UB, all of Rust's normal guarantees still apply to the safe parts. Because of this, we feel that Rust still keeps much of its safety guarantees even in the domain of lock-free concurrent programming, and thus it should be a

major step up in safety compared to C/C++.

6.1.3 Performance

Under a pairwise workload, the throughput as the concurrency level rises for both the MS queue and LPRQ implementation is what we believe to be most in line with what could be expected. However, even though this workload puts a lot of pressure on the queue, it is far from a real-world scenario. As mentioned, the MPMC workload, especially the 2:1 ratio, is the closest to a practical scenario.

For the 2:1 ratio results, we can determine that Rust with hazard pointers is a contender to C++ when comparing the results at the highest level of concurrency, especially when it comes to memory access. However, Rust's inconsistent throughput as concurrency increases is unexpected and notable.

The most unexpected results are the ones from the 1:1 ratio. In this workload, the Rust versions performed significantly worse than the C++ version once more than a few threads were used. As it seems to affect both the hazard pointer and epoch version in a similar way we have two theories. Either the issue lies in our LPRQ implementation, or it is caused by something more in the Rust language or its standard library. Either way, it could also be what is causing the unexpected results in the 2:1 ratio. Despite significant time spent profiling trying to pinpoint the cause, we did not manage to come to any definite conclusion. Compliant with our methodology, the code in each language is very similar, and upon inspecting the generated assembly we could not find any significant difference for many of the performance-critical parts.

Furthermore, as we rely on existing libraries for memory reclamation, a comparison between these across languages might also be necessary to know with more certainty to what degree they contribute to or decrease performance. However, with the results we have from the three different versions, hazard pointers seem to be the most efficient technique, which is perhaps not a surprise as it is the advocated solution.

6.1.4 Further research

Further analysis into this area could be to perform a comparison of how each language transforms its code into assembly. Such an analysis could provide a deeper understanding of the core differences the languages pose against each other. Such a mapping could contribute to improving the overall performance of Rust by finding inefficiencies in the Rust compiler.

Investigating Rust's memory usage and access behavior could also be interesting. Though it might be isolated to our specific implementation, the results indicate that Rust utilizes the stack heavily compared to C++ and our best assumption is Rust's standard library causes that. Additionally, as it is also apparent in the more idiomatic Rust version with `aarc`, such an investigation could provide insight into why Rust sometimes induces a large number of page faults compared to C++.

Another relevant direction could be to expand on the performance of memory reclamation techniques for concurrent lock-free data structures. This could build upon previous research that implemented a high-performance GC in Rust [24] and compare it with other solutions.

Even though we believe we include a variation of measurements that together provide a robust overview of each language's performance, measuring throughput under different levels of congestion would also have been interesting. As our approach is to calculate the throughput from the total runtime it did not make sense to introduce congestion. From our understanding, forcing lower congestion means that threads have to be delayed in some sense, only resulting in a longer runtime for our approach, which would be a contradiction to the intuition that lower congestion should give higher throughput. This together with time constraints we, unfortunately, delegate to future studies.

Additional metrics related to the inner workings of the data structure could also be of interest when comparing an implementation in different languages. For the LPRQ this means gathering information about the total number of PRQs that were allocated during the benchmark, average time spent in enqueue, and dequeue operations, and failed CAS operations, as well as total executed atomic operations. Though implementation-specific, these fine-grained metrics could provide insight into potential performance bottlenecks which could contribute to understanding the broader measurements.

6.2 Conclusion

In this thesis, we have implemented two concurrent lock-free FIFO queue algorithms in Rust to assess whether Rust is suitable for high-performance concurrent lock-free programming. More specifically we compared our implementation of the classic MS queue in Rust with an existing version in C, and our Rust version of the LPRQ algorithm with its counterpart in C++. In our quantitative, performance, comparison, we included metrics such as throughput, energy consumption, memory usage, and memory access pattern. For our qualitative comparison, we assessed the programming ergonomics provided by Rust and C++. Additionally, we reasoned about how Rust's safety guarantees are affected by the necessary use of unsafe code for our implementation.

Promisingly, in the most realistic scenario, the MPMC workload with a 2:1 ratio, Rust can compete with C++ in performance. Even under the pairwise workload, the LPRQ implementation shows comparable throughput. Furthermore, although perhaps not as important as the 2:1 ratio, as a perfectly balanced load is not a practical example, the results from the 1:1 ratio for the MPMC workload were confusing and left us only with theories as to why. Further investigation into the generated assembly code, as well as gathering metrics about the internal workings of the LPRQ could provide some answers.

Moreover, we acknowledge the limitations of our decision to rely on an external benchmarking tool which resulted in less control over the benchmarking process.

This unfortunately left us with a degree of uncertainty about some parts of our results. However, given that all implementations were executed under identical conditions, this may have mitigated the overall impact.

Interestingly, the idiomatic Rust version, even though worst in performance, preserves Rust's safety guarantees and could therefore have its practical use cases, but perhaps far away from production code. For the other implementations, although not available in stable Rust, the powerful Miri still provides some safety to unsafe code. This allows us to some degree be confident that unsafe code is not as unsafe as the name suggests.

Finally, Rust's ability to match or even surpass C++ in performance, combined with its user-friendly features, shows it has a promising future. We believe that Rust will continue to increase in popularity, and can be considered as an option for high-performance concurrent lock-free programming.

Bibliography

[1] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. "Concurrent deferred reference counting with constant-time overhead". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 526–541. ISBN: 9781450383912. DOI: 10.1145/3453483.3454060. URL: https://doi.org/10.1145/3453483.3454060.

[2] S Tanenbaum Andrew and Bos Herbert. *Modern operating systems*. Pearson Education, 2015.

[3] Vytautas Astrauskas et al. "How do programmers use unsafe rust?" In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–27.

[4] Aras Atalar et al. "Modeling Energy Consumption of Lock-Free Queue Implementations". In: *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 2015, pp. 229–238. DOI: 10.1109/IPDPS.2015.31. URL: https://doi.org/10.1109/IPDPS.2015.31.

[5] Alexis Kenneth Beingessner. "You Can't Spell Trust Without Rust". PhD thesis. Carleton University, 2016.

[6] Randal E Bryant and David Richard OHallaron. *Computer systems: a programmers perspective*. Prentice Hall, 2011.

[7] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[8] Crossbeam. *Crossbeam crate*. 2024. URL: https://docs.rs/crossbeam/latest/crossbeam/.

[9] Crossbeam. *crossbeam::utils::CachePadded*. 2024. URL: https://docs.rs/crossbeam/latest/crossbeam/utils/struct.CachePadded.html.

[10] Ulrich Drepper. "What every programmer should know about memory (2007)". In: *URL http://people. redhat. com/drepper/cpumemory. pdf* (2010).

[11] Rust Foundation. *Data Races and Race Conditions - The Rustonomicon*. URL: https://doc.rust-lang.org/nomicon/races.html. 2023.

[12] Keir Fraser. *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[13] Jon Gjengset. *haphazard - A Rust library for hazard pointers*. 2024. URL: https://crates.io/crates/haphazard/0.1.8/dependencies.

[14] Danny Hendler et al. "Flat combining and the synchronization-parallelism tradeoff". In: *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 2010, pp. 355–364.

[15] Maurice Herlihy et al. *The art of multiprocessor programming*. Newnes, 2020.
[16] Moshe Hoffman, Ori Shalev, and Nir Shavit. "The baskets queue". In: *Principles of Distributed Systems: 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings 11*. Springer. 2007, pp. 401–414.
[17] Lilly Jinstrand and Marcus Julin. *Lock-Free-Concurrency-in-Rust-Analysis-of-Rust-for-Lock-Free-Programming*. `https://github.com/mmackan/Lock-Free-Concurrency-in-Rust-Analysis-of-Rust-for-Lock-Free-Programming`. 2024.
[18] Ralf Jung. *Miri*. 2024. URL: `https://github.com/rust-lang/miri`.
[19] Ralf Jung. "Understanding and evolving the Rust programming language". In: (2020).
[20] Ralf Jung et al. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.
[21] Ralf Jung et al. "Safe systems programming in Rust". In: *Communications of the ACM* 64.4 (2021), pp. 144–152.
[22] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. `https://doc.rust-lang.org/stable/book/`. Accessed: 2024-03-01.
[23] Edya Ladan-Mozes and Nir Shavit. "An optimistic approach to lock-free FIFO queues". In: *Distributed Computing: 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004. Proceedings 18*. Springer. 2004, pp. 117–131.
[24] Yi Lin et al. "Rust as a language for high performance GC implementation". In: *ACM SIGPLAN Notices* 51.11 (2016), pp. 89–98.
[25] Linux Kernel Organization. *perf: Linux profiling with performance counters*. URL: `https://perf.wiki.kernel.org/index.php/Main_Page`.
[26] Paul E McKenney. "Is Parallel Programming Hard, And, If So, What Can You Do About It?(Release v2023. 06.11 a)". In: *arXiv preprint arXiv:1701.00854* (2017).
[27] Memusage command. *Man page for memusage command*. URL: `https://man7.org/linux/man-pages/man1/memusage.1.html`.
[28] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects". In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (2004), pp. 491–504.
[29] Maged M Michael and Michael L Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996, pp. 267–275.
[30] Adam Morrison and Yehuda Afek. "Fast concurrent queues for x86 processors". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 103–112.
[31] David A Patterson and John L Hennessy. *Computer organization and design: the hardware software interface*. Morgan kaufmann, 2013.

[32] Rui Pereira et al. “Energy efficiency across programming languages: how do energy, time, and memory relate?” In: *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*. 2017, pp. 256–267.

[33] David Peter. *hyperfine*. Version 1.16.1. Mar. 2023. URL: `https://github.com/sharkdp/hyperfine`.

[34] Boqin Qin et al. “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.

[35] Baishakhi Ray et al. “A large scale study of programming languages and code quality in github”. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 2014, pp. 155–165.

[36] Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science & Business Media, 2012.

[37] Raed Romanov and Nikita Koval. *LPRQ Concurrent Queue Algorithm Evaluation*. `https://zenodo.org/record/7337237`. Version 1.0. Source code of the experiments presented in the paper "The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2". 2023.

[38] Raed Romanov and Nikita 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*. 2023, pp. 14–26.

[39] Michael Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.

[40] Stack Overflow. *2023 Developer Survey*. 2023. URL: `https://survey.stackoverflow.co/2023/`.

[41] Laszlo Szekeres et al. “Sok: Eternal war in memory”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.

[42] The Rust Team. *core::arch::x86_64::cmpxchg16b - Rust*. `https://doc.rust-lang.org/core/arch/x86_64/fn.cmpxchg16b.html`. Accessed: 2024-03-18.

[43] The Rust Team. *Learn Rust*. 2024. URL: `https://www.rust-lang.org`.

[44] The Rust Team. *Rust Standard Library Documentation*. 2024. URL: `https://doc.rust-lang.org/std/`.

[45] The Rust Team. *Rust: Production*. 2024. URL: `https://www.rust-lang.org/production`.

[46] The Rust team. *Rustdoc Unstable Features*. 2024. URL: `https://doc.rust-lang.org/rustdoc/unstable-features.html`.

[47] The Rust team. *std::ptr - Strict Provenance*. 2024. URL: `https://doc.rust-lang.org/nightly/std/ptr/index.html#strict-provenance`.

[48] The Rust team. *The Rust Reference*. 2024. URL: `https://doc.rust-lang.org/reference/type-layout.html#representations`.

[49] Brian L Troutwine. *Hands-On Concurrency with Rust: Confidently build memory-safe, parallel, and efficient software in Rust*. Packt Publishing Ltd, 2018.

[50] Philippas Tsigas and Yi 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*. 2001, pp. 134–143.

[51] wyang5. *AARC Docs.rs*. URL: https://docs.rs/aarc/0.2.1/aarc/index.html.

[52] Chaoran Yang. *Fast Wait Free Queue*. https://github.com/chaoran/fast-wait-free-queue.

[53] Joshua Yanovski et al. "Ghostcell: separating permissions from data in rust". In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–30.

[54] Zeming Yu, Linhai Song, and Yiying Zhang. "Fearless concurrency? understanding concurrent programming safety in real-world rust software". In: *arXiv preprint arXiv:1902.01906* (2019).

A

Values from MPMC benchmarks

Table A.1: Raw values for MPMC 1:1 benchmarks.

Implementation	**cache-references**	**cache-misses**	**faults**	**heap peak**	**stack peak**	**energy (J)**	**time (s)**
C++	3833693336	239887	1321	92289	10272	1259.34	8.68
Hazp	4376046737	528825	19715	46597	263904	2273.19	15.63
Aarc	7143515747	621605	11887	84413	271744	3979.93	29.0
Epoch	2794588435	594187	23010	80197	275424	1985.56	13.4

Table A.2: Raw values for MPMC 2:1 benchmarks.

Implementation	**cache-references**	**cache-misses**	**faults**	**heap peak**	**stack peak**	**energy (J)**	**time (s)**
C++	3588122034	23203783	366869	92353	10272	1085.23	7.5
Hazp	2380371834	1158281	75529	67237	263904	1055.51	7.53
Aarc	21094219295	26316609	2498097	112581	271728	2815.19	18.71