University of Stuttgart
Institute for Control Engineering
of Machine Tools and Manufacturing Units
(ISW)

Master's Thesis

Wait-free synchronisation for inter-process communication in real-time systems

submitted by

Devrim Baran Demir

from Hamburg

Degree program	M. Sc. Informatik
Examined by	Prof. Andreas Wortmann
Supervised by	Marc Fischer
Submitted on	July 16, 2025

Declaration of Originality

Master's Thesis of Devrim Baran Demir (M. Sc. Informatik)

Address	Kernerweg 22, 89520 Heidenheim an der Brenz
Student number	3310700
English title	*Wait-free synchronisation for inter-process communication in real-time systems*
German title	*Wait-free Synchronisation für Interprozesskommunikation in Echtzeitsystemen*

I now declare,

- that I wrote this work independently,
- that no sources other than those stated are used and that all statements taken from other works—directly or figuratively—are marked as such,
- that the work submitted was not the subject of any other examination procedure, either in its entirety or in substantial parts,
- that I have not published the work in whole or in part, and
- that my work does not violate any rights of third parties and that I exempt the University against any claims of third parties.

Stuttgart, July 16, 2025

Kurzfassung

Vorhersehbare und korrekte Interprozesskommunikation (IPC) ist für Echtzeitsysteme von entscheidender Bedeutung, da Verzögerungen, Unvorhersehbarkeit oder inkonsistente Datenstände zu Instabilität und Ausfällen führen können. Traditionelle Synchronisationsmechanismen verursachen Blockierungen, die zu Prioritätsinversionen und erhöhten Antwortzeiten führen. Um diese Herausforderungen zu bewältigen, bietet die wartefreie Synchronisation eine Alternative, die den Abschluss von, wie der Austausch von Daten zwischen mehreren Prozessen, in einer begrenzten Anzahl von Schritten garantiert und so die Systemreaktionsfähigkeit und -zuverlässigkeit sicherstellt.

Diese Arbeit untersucht die Nutzung von wait-free Datenstrukturen für IPC in Echtzeitsystemen mit Fokus auf deren Implementierung in Rust. Das Eigentumsmodell und die strengen Nebenläufigkeitsgarantien von Rust machen es besonders geeignet für die Entwicklung latenzarmer und hochzuverlässiger Synchronisationsmechanismen. Diese Arbeit analysiert bestehende wait-free Methoden für IPC in Echtzeitsystemen und bewertet ihre Leistung im Vergleich zu herkömmlichen Synchronisationsmethoden.

Stichwörter: Echtzeitsysteme, wait-free Sysnchronisation, lock-free Synchronisation, Interprozesskommunikation, rust

Abstract

Predictable and correct Inter-Process Communication (IPC) is essential for Real-Time System (RTS), where delays, unpredictability or inconsistent data can lead to instability and failures. Traditional synchronization mechanisms introduce blocking, leading to priority inversion and increased response times. To overcome these challenges, wait-free synchronization provides an alternative that guarantees operation completion, such as the completion of data exchange between multiple processes, within a bounded number of steps, ensuring system responsiveness and reliability.

This thesis explores the use of wait-free data structures for IPC in RTS, focusing on their implementation in Rust. Rust's ownership model and strict concurrency guarantees make it well-suited for developing low-latency and high-reliability synchronization mechanisms. This work examines existing wait-free techniques for real-time IPC, and evaluates their performance against conventional synchronization methods.

Keywords: real-time systems, wait-free synchronization, lock-free synchronization, inter-process communication, rust

Contents

1. Introduction

1.1. Motivation

In modern manufacturing and automation, control systems must operate under strict timing constraints to function reliably. If a system fails to meet these constraints, unexpected delays can disrupt processes, leading to instability or even hazardous failures in safety-critical environments. For this reason, RTS and low-level programming languages like C or Rust are widely used to ensure predictable execution times.

To achieve these strict timing requirements, many real-time applications involve multiple tasks that must run concurrently and share resources efficiently. Without proper synchronization, problems such as data corruption or race conditions can occur leading to unpredictable behavior. Traditional synchronization methods with locks are commonly used to manage access to shared resources by blocking processes so that only one process at time accesses the shared resource to exchange data in a proper way. However, these blocking mechanisms introduce significant challenges in real-time settings. Since Traditional synchronization methods require processes to wait for resource availability, they can lead to increased response times, potential deadlocks, potential process starvations, and potential priority inversions. These delays are unacceptable in systems that require strict timing guarantees. [1]–[3]

To overcome these limitations, there is an increasing interest in synchronization techniques without any blocking mechanisms. A lock-free algorithm for instance functions withouth any locking mechanism thus no blocking. This guarantees that at least one process completes in a finite number of steps, regardless of contention (multiple processes try to access the same shared resource). This property ensures that at least the system will still work eventhough one process might be lagging. The only problem is that this will not handele starvation since there is no guarantee that every process will finish its task. [4]

While lock-free algorithms represent a significant improvement, wait-free algorithm guarantees that every operation completes in a finite number of steps, regardless of contention. This property ensures system responsiveness and predictability, which are essential for real-time applications. By eliminating blocking and contention-based delays, wait-free synchronization prevents priority inversion and ensures that high-priority tasks execute without interference. [1], [2], [4]

These synchronization mechanisms are particularly important in the context of IPC, which plays a crucial role in RTS. IPC allows processes to exchange data efficiently, but its performance is heavily influenced by the synchronization techniques used. Traditional IPC mechanisms, which often rely on blocking some processes, can introduce significant latency and reduce throughput. Wait-free data structures offer a promising alternative by ensuring that communi-

cation operations complete within predictable time bounds. However, selecting appropriate wait-free data structures and evaluating their performance in real-time environments remains a challenge. [5]–[8]

To implement these advanced synchronization techniques safely, the choice of programming language is crucial. The Rust programming language provides useful features for implementing real-time synchronization mechanisms. Its ownership model and strict type system prevents data races and enforce safe concurrency. Additionally, Rust offers fine-grained control over system resources, making it a strong candidate for real-time applications that demand both low latency and high reliability. [9], [10]

The concepts and methods introduced here, including RTS, IPC, synchronization techniques and problems, wait-free synchronization, and the rust programming language are explored in greater depth in chapter 2.

1.2. Objective

The primary goal of this research is to find wait-free data structures that can be used to implement a wait-free synchronization for IPC though shared memory in RTS using Rust. To do so, this study aims to:

- Identify and analyze existing wait-free synchronization techniques for IPC through shared memory for RTS.
- Implement and compare the performance of existing wait-free synchronization mechanisms for IPC through a shared memory for real-time scenarios with each other.
- Choose and analyze which wait-free data structure for IPC through shared memory in a real-time setting using Rust that is best suited.

By addressing these objectives, this work contributes to the field of wait-free synchronization for IPC in RTS by providing a practical solution with rust. The insights gained from this research can help improve the reliability and performance of real-time applications across various domains.

1.3. Structure of the Thesis

2. Background

To establish a clear foundation for the concepts and definitions introduced throughout this thesis, a fundamental overview of the key topics relevant to this research will be provided. This includes an introduction to RTS, Inter-Process Communication (IPC), and synchronization techniques, with a particular focus on wait-free synchronization. Additionally, the Rust programming language will be examined, as it serves as the primary development environment for this study. Furthermore, existing synchronization methods in RTS will be explored to contextualize the motivation and contributions of this work.

2.1. Real-Time Systems

In RTS the correctness of the system does not only depend on the logical results of computations, but also on timing constraints. These systems can be classified into Hard Real-Time System (HRTS) or Soft Real-Time System (SRTS). HRTS have strict timing constraints, and missing a constraint is considered a system failure and may lead to a catastrophic desaster. The sysetem must guarantee that every timing constraint has to be met. An use case would be industrial automation where all the machines and robotic modules have to communicate with each other as quick as possible to ensure no blockage of the manufacturing line. [11]

On the other hand, SRTS try to stick to the timing constraints as much as possible, but missing some timing constraints is not considered a system failure. Infrastructure wise SRTS are similar to HRTS, since it is still considered important to meet these timing constraints. An example would be a multimedia system where it would be considered fine if sometimes frames are dropped to guarantee the video stream. [11]

Sometimes these two systems appear in combination, where some functions have hard real-time constraints and some have soft real-time constraints. Krishna K. gives a good example in his paper where he describes that for the apollo 11 mission some components for the landing processes had soft real-time behavior and the rest still functioned with hard real-time constraints. [11]

Since the workfield of this thesis is within HRTS, the term RTS will be used synonymously with the terminology HRTS.

2.2. Inter-Process Communication

Processes used in a RTS also have to share information with each other so the system can function. So some kind of IPC is needed. IPC allows processes to share information with each

other using different kind of methods like a shared memory region, which will be the method used and explained later in this thesis. In general, IPC is needed in all computing systems, because processes often need to work together (e.g. a producer process passes data to a consumer process). Lets take the brake-by-wire technology as example. Brake-by-wire is a technology for driverless cars where some mechanical and hydraulic components from the braking systems are replaced by wires to transmit braking signals, since there is no driver anymore to press on the braking pedal [12]. This of course requires different processes to share information together. In the context of this thesis this kind of communication requires strict timing constraints as stated as before, since any kind of delay or blockage would lead to fatal consequences. [13], [14]

2.2.1. Shared Memory

To achieve any kind of information sharing between processes, these processes will need to have access to the same data regularly. With a shared memory segment, multiple processes can have access to the same memory location. So all processes which are part of the IPC can read and write to this common memory space avoiding unnecessary data copys. With that, processes exchange information by directly manipulating memory. This kind of IPC is particular useful for real-time applications, which handle large volumes of data or are required to quickly transfer data between sensors and control tasks. What is also important to know is that the section of the code, that programs these data accesses by different processes is called critical section. The problem with this is that the system somehow has to manage how the processes access the shared memory segment. This is mostly done by using different kind of synchronization techniques. Without any synchronization mechanism race conditions or inconsistent data can occur. [15]

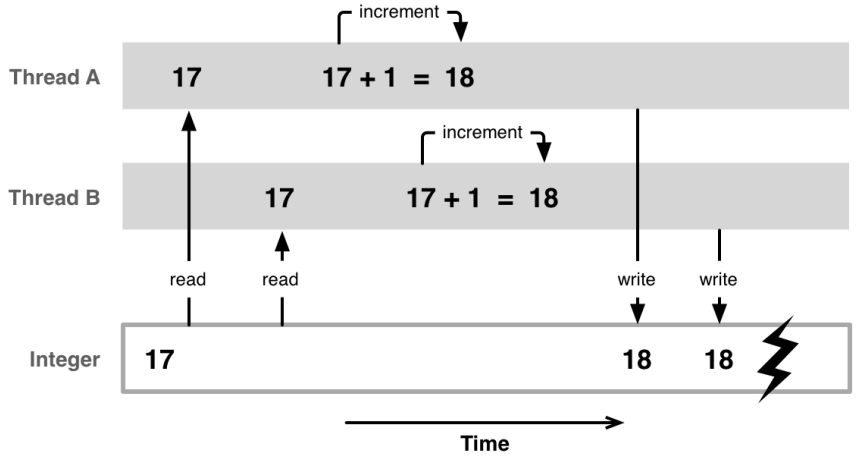
2.3. Synchronization

As observed, synchronization is a crucial part of IPC in RTS, especially when processes communicate via shared memory. Communication through shared memory always has a risk of race conditions and data inconsistency if the processes are not properly synchronized. Tratitional synchronization techniques ensure mutual exclusion (only one process at a time uses shared resource) thus avoiding race conditions and ensuring data consistency. Race conditions happen when for example two processes write to the same resource. Lets take a single counter instance with value 17 as a shared resource in a shared memory region. If one has process p1 and one process p2 increments that number, the end result should be 19. But what could happen is that p1 could read the value 17 before p2 increments it and then before p1 increments that value p2 could also read the value 17. Now internally both processes increment that number to 18 and both processes would write 18 to that shared resource. To understand this example more in detail fig. 2.1 visualizes a race condition with threads.

The difference between processes and threads is just, that threads are part of a process which can perform multiple tasks via threads simultaneously within that process. Another

difference that will later be important in this thesis is that processes have their own private memory space, while threads share the memory space of the process they are part of. So naturally a process can not access the memory of another process. Regardless the following concept in fig. 2.1 can be used for processes too. [16]

Figure 2.1.: Race condition between two threads, which write to the same shared variable.



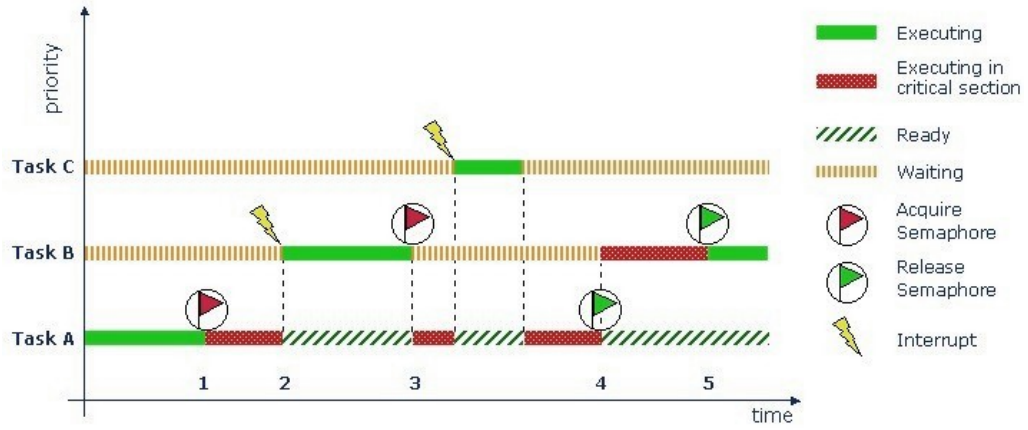
[17]

2.3.1. Mutual Exclusion

As discussed mutual exclusion does only allow one process or thread to access the shared resource at a time. This includes that if a process p1 already accessed the shared resource x and is still working on it, a second process p2, which trys to access that shared resource x has to wait untill the process p1 finishes its task, where it needs that shared resource x. To achieve this mostly synchronisation techniques based on locks or semaphores are used to block the entry of an process to an already accessed and in use shared resource of an other process. See fig. 2.2 to gain an deeper understanding on how this works. This paper will not go into detail how traditional synchronisation techniques like locks or semaphores work, since for this work it is only important that these kind of methods manage the access of processes to shared resources in shared memorys via some kind of locks. A process will aquire a lock to access a shared resource and will release it when its task is done. Another process trying to access the same resource while its in use has to wait untill the lock is released for that resource.

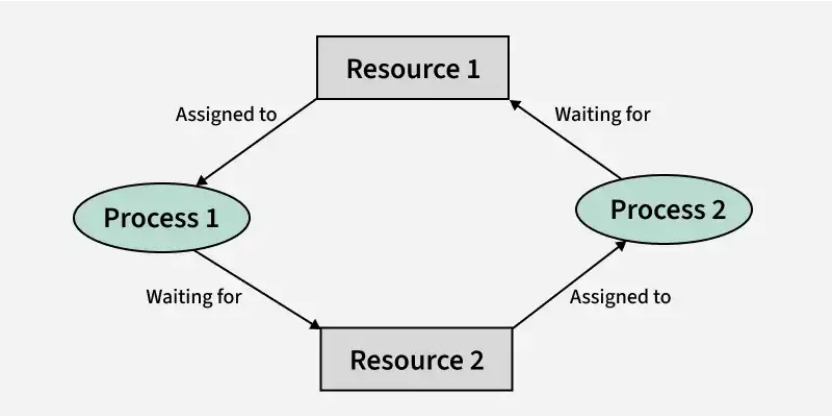
It is clear that this approach inherently relys on blocking a set of processes. This may lead to several issues, including deadlocks, process starvation, priority inversion, and increased response times. The sequence which process might aquire the lock first to enter the critical section, when multiple processes wait for the access is mostly set by a scheduler. Since wait-free methods explained later in section 2.5 are lock-free, a scheduler is not needed and as a result of that scheduling will not be explained more in detail in this work. [2], [19]

Figure 2.2.: Mutual exclusion between three tasks(processes), which access the same critical section. Multiple processes need to stop working and just wait for other processes to finish their work. See the waiting phase of the processes.



[18]

Figure 2.3.: Deadlock between two processes, which wait for each other to release the needed resources.



[20]

Process Starvation

Another problem would be, what if when multiple processes try to enter the shared resource one after the other and one process keeps getting outperformed in aquiring a lock to enter the critical section. This process would wait for a indefinite time and will never enter the shared resource. A process that will never access a shared resource that it trys to enter starves out. This usually happens when a synchronization method allows one or more processes to make progress while ignoring a particular process or processes. This mostly happens in environments where some sort of process prioritization exists and processes are classified into low and high priority processes. When there are always high priority processes available and

some low priority processes, it might happen that these low priority processes will never be able to enter the critical section. This is a problem, since these low priority processes might be important for the system too. [21]

Deadlock

Even worse, what if two or more processes already accessed a resource and now each of them wait that the other process releases the lock for the resource each of them aquired? This results in a situation seen in fig. 2.3 where these processes now indefinetlywait for each other and never terminate. So the resources that these waiting processes hold are also never released and are also never available to other processes maybe needed by other systems. As one can see, this a system into a state which would make no progress any further and would also not respond to any command anymore. [20], [22]

For instance a driverless car with a brake-by-wire system, where processes responsible for braking are in a deadlock, could eventually not brake if needed and a fatal car crash would happen.

Priority Inversion

Now lets say no process starvations or deadlocks happen. What could happen too is that a lower priority process already accessed a shared resource and after that a higher prioritiy process needs to access that specific resource too. If the lower priority process now gets delayed, the higher priority process gets delayed too. This would be called priority inversion now; a low priority process delaying an high prioritiy process. [23]

Increased Response Times

As demonstrated traditional synchronization is based on mutual exclusion via blocking processes. The result is processes that are waiting, which leads to increased response times of a system. This results in not meeting the timing constraints of a HRTS environment.

2.4. Lock-Free Synchronization

Therefore synchronization techniques are required that do not block processes with any kind of locking mechanism. One way could be the implementation of lock-free syschronization techniques. This would allow multiple processes to access the shared resource concurrently. Lock-free synchronization ensures that at least one process will make progress in a finite number of steps. However some processes may be unable to proceed, because lock-free synchronization does not guarantee that all processes will complete their operations in a finite number of steps. This means that starvation or even prioritiy inversion is still possible, as some processes, even high priority processes may be indefinitely delayed by others. There are different kind of mechanisms to achieve this. One way to accomplish lock-freedom for

example is the lock-free technique introduced by Michael and Scott, which is also the basis for some other wait-free algorithms.

2.4.1. Michael and Scott's Lock-Free Queue

Michael and Scott developed an algorithm seen in algorithm 1 using a linked-list as a shared data structure with a enqueue and a dequeue function to introduce lock-freedom. A linked-list is a list containing nodes containing data and a pointer called next which references the next node in the list, which can only be traversed in a single direction. There is also a pointer called head, which references the beginning node of the list and a pointer calles tail, which references the end node of the list. The core concept of the algorithm is the enqueue and dequeue functions beginning at line 7 and 26 in algorithm 1, which are used to add and remove nodes to the shared data structure. When a process trys to add a node to the list, it first creates a new node and sets its next pointer to NULL, see line 8 to 10 of the enque function. Beginning from line 11 to line 23 following happens: The process first checks if the pointer referencing to the next node after the tail node is NULL, see line 15. If it is null it tries to link the new node to the end of the list by using a Compare and Swap (CAS) seen in line 16. This operation atomically compares the current value of the tail pointer with the expected value and, if they match, updates the tail pointer to point to the new node. The tail itself would be updated in line 24. [24]

So lets say 2 processes p1 and p2 untill line 16 executes one after the other. What could happen now is that if p1 executes line 16 before p2, p2 will fail the CAS from line 16. Now if p1 would not execute further thus not finalizing the enqueue with line 24 and p2 retrys the loop until line 15, the condition in line 15 would not be TRUE anymore for p1 and p1 would execute line 19 and 20 to help p2 to finalize its enqueue so other processes can work further with this algorithm. [24]

The dequeue function works analogously, but instead of adding a node to the end of the list it removes a node from the front of the list. And since another process which could not finish its enqueue would cause confusion for other processes in the dequeue function, the process which could not finish its enqueue will also be helped in the dequeue function. [24]

Initialization starts at line 1 in algorithm 1, which is just used to to create dummy nodes when there's no node in the list. This just simplifies the algorithm so that the head and tail pointers are not null. It can be observed that this approach does not need any locks explained in section 2.3. Still this approach has one major problem. If for instance process p1 is trying to enqueue, it can happen that the CAS loop might fail indefinetly if for an indefinite time other processes are always executing line 16 immediately befor p1 could execute line 16. This means that in very high contention scenarios, a process may be delayed indefinetly and starve out. In a HRTS this could lead to violating timing constraints, since the process would not finish his task in the defined timing window, which is unacceptable. This is why a slightly different approach which guerantees that every process will complete its operation in a finite number of steps is in need. [24]

Algorithm 1 Michael and Scott's Lock-Free Queue

```
1:  function INITIALIZE(Q : pointer to queue_t)
2:      node = new node()                                   ▷ Allocate a dummy node
3:      node.next.ptr = NULL                                ▷ Make it the only node in the list
4:      Q.Head = node                                       ▷ Both Head and Tail point
5:      Q.Tail = node                                       ▷ to this dummy node
6:  end function
7:  function ENQUEUE(Q : pointer to queue_t, value : data_type)
8:      node = new node()                                   ▷ Allocate a new node from the free list
9:      node.value = value                                  ▷ Copy enqueue value into node
10:     node.next.ptr = NULL                                ▷ Set next pointer of node to NULL
11:     loop                                                ▷ Keep trying until Enqueue is done
12:         tail = Q.Tail                                   ▷ Read Tail (pointer + count) together
13:         next = tail.ptr.next                            ▷ Read next ptr + count together
14:         if tail == Q.Tail then                          ▷ Are tail & next consistent?
15:             if next.ptr == NULL then                    ▷ Tail is the last node?
16:                 if CAS(&tail.ptr.next, next, ⟨node, next.count + 1⟩) then
17:                     break                               ▷ Link the new node; Enqueue is done
18:                 end if
19:             else                                        ▷ Tail not pointing to the last node
20:                 CAS(&Q.Tail, tail, ⟨next.ptr, tail.count+1⟩)   ▷ Move Tail forward (helping another enqueuer)
21:             end if
22:         end if
23:     end loop
24:     CAS(&Q.Tail, tail, ⟨node, tail.count + 1⟩)          ▷ Final attempt to swing Tail to the inserted node
25: end function
26: function DEQUEUE(Q : pointer to queue_t, pvalue : pointer to data_type)
27:     loop                                                ▷ Keep trying until Dequeue is done
28:         head = Q.Head
29:         tail = Q.Tail
30:         next = head.ptr.next                            ▷ Read head->next
31:         if head == Q.Head then                          ▷ Still consistent?
32:             if head.ptr == tail.ptr then                ▷ Empty or Tail behind?
33:                 if next.ptr == NULL then                ▷ Queue is empty
34:                     return FALSE
35:                 else                                    ▷ Tail is behind, help move it
36:                     CAS(&Q.Tail, tail, ⟨next.ptr, tail.count + 1⟩)
37:                 end if
38:             else                                        ▷ No need to adjust Tail
39:                 *pvalue = next.ptr.value                ▷ Read value before CAS
40:                 if CAS(&Q.Head, head, ⟨next.ptr, head.count + 1⟩) then
41:                     break                               ▷ Dequeue is done
42:                 end if
43:             end if
44:         end if
45:     end loop
46:     free(head.ptr)                                      ▷ Safe to free old dummy node
47:     return TRUE
48: end function
```

[24]

While Michael and Scott's algorithm relies on the CAS primitive, other atomic primitives provide alternative approaches that other algorithms shown later in this thesis use. A overview on the atomic primitives that are used in this thesis context is given in the following section.

2.4.2. Atomic Primitives

Atomic primitives are hardware instructions that conduct a set of steps atomically, meaning with no interruption from other processes [25]. This will be important in the algorithms analyzed later in this thesis, since these primitives are used to implement wait-free synchronization. There are different kind of atomic primitives:

Load-Linked and Store-Conditional (LL/SC)

Abbreviation of the instructions Load-Linked (LL) and Store-Conditional (SC), which is an operation available on ARM, MIPS and Alpha architecutures usually implied with a Validate-Link (VL) instruction.

- LL(R) returns value of register r
- "SC(R, v) changes the value in register R to v and returns true, if and only if no other process performed a successful SC since the most recent call of LL of the current process. So SC fails if the value of the register has changed since it has been read" [26]
- "VL(R) returns true if no other process performed a successful SC on register R, which allows to test a register value without changing it" [26]

[26]

Compare and Swap (CAS)

It is already explained how CAS works in section 2.4.1. Furthermore to the explanation in section 2.4.1 CAS is an atomic primitive that is supported on "intel x386, x64 and most general purpose architectures with operands that are restricted to pointer size" [26].

- "CAS(R,e,n) returns true and sets the value of R to n if the value in R is e. Otherwise, it returns false." [26]

The problem with CAS furthermore than the issue explained earlier in section 2.4.1 is that it can lead to the ABA problem, which can also occur in wait-free algorithms:

- Process 1 reads value A from a shared variable.
- Process 2 changes the value to B and then back to A.
- Process 1's CAS operation succeeds, because it compares the value A it read earlier with the current value A, even though the value was changed in between.

This is a fundemental limitation of CAS. One solution would be to replace CAS with LL/SC, but that is not possible on x86 processors. So other solutions are needed that are discussed in **??**. [26]

Double-With Compare and Swap (DWCAS)

This is a CAS on two neighboring memory locations. [26]

Double Compare and Swap (DCAS)

Sometimes also called CAS2, is a CAS on two independent memory locations. [26]

Swap

Swap is an atomic read-modify-write operation that unconditionally exchanges a value in memory with a new value and returns the old value. Swap(R, v) atomically stores value v into location R and returns the previous value that was in R. This operation always succeeds. [27].

Fetch and Add (FAA)

This primitive is used to increment "the value of a variable by a given offset and [return] the result. This instruction always succeeds." [26]

Fetch and Store (FAS)

This atomically stores a value into a variable and returns the previous value. This is similar to CAS, but it does not require a comparison and a retry loop. This is faster then CAS, if conditions before updating do not need to be checked. [28]

2.5. Wait-Free Synchronization

Lock-freedom solves the problem off a system getting into a deadlock. But this is not enough, in a fully automated car for example it is undesirable that any process does not complete its task, since that could mean that some processes that are responsible for braking would not finish their work in a worst case scenario. And such an occasion where the car would need to brake a fatal car crash would be the outcome. Consequently a solution is necessary where every process finishes their task in a finite number of steps instead of just one process. So something is needed which builds upon such mechanisms and expand them. This is exactly what wait-free synchronization is. It guarantees that every process will complete its operation in a finite number of steps, regardless of contention. This means that even process starvation is by definition not possible anymore. So by definition starvation cannot happen anymore. Also priority inversion is eliminated too, because processes do not have to wait for other processes anymore. This ensures system responsiveness and predictability thus the ability to define strict timing constraints to meet these vital timing constraints, which are essential for HRTS applications. But even wait-free algorithms introduce one problem. Wait-free algorithms are in most cases slower than their lock-free counterpart in execution. A solution to this problem was the fast-path / slow path method by Kogan and Petrank, where operations on a data

structure is first done with a bounded lock-free operation, and when failed a slower wait-free operation will substitute the failed lock-free operation [4]. This will be analysed more in detail in chapter 5.

2.6. Rust Programming Language

The question now is which programming language suits best for this kind of research. Since a fast communication between the processes is compulsory to meet all HRTS timing constraints, the C programming language would be a good choice. C provides low-hardware control and therefore also allows the implementation of fast low-latency communication. What is also important and necessary for a RTS is, that C does not have an automatic garbage collecter, which gets active and stops all processes from working to clean up allocated but no longer used memory space. Because of that all Real-Time Operating System (RTOS) are written in C. The main problem with C is that it does not provide any kind of memory safety, since C implements memory operations that are prone to buffer overflows or control-flow attacks. In the industry around 70% of vulnarabilities happen because of memory safety issues. If the real-time application would run on an isolated system with no internet connection, this would not be a problem. But in modern automation, where systems need to be connected to the internet for data exchange, such systems would be prone to security attacks. RTS nowadays is an integral part in various connected devices, including critical fields such as health or transportation. Conclusively it is extremely important that the security of such devices is guaranteed. With the rust programming language the problems of memory safety features are gone. The difference to C is that it can be as fast as C with the possibility to support low-level control and high-level programming features, while providing memory safety features in a real-time setting. The memory safety aspect is achieved by an ownership concept that controls how memory is handled in programs. This is strictly checked and therefore the executable program has guaranteed memory safety. In the model every value has a single owner represented by a variable. The owner is in charge of the lifetime and deallocation of that value. Rust will automatically free the memory associated with that value, when the owner goes out of scope. This behaviour is automatically done by using the memory reference feature provided by rust. Creating such references is also called borrowing. This allows the usage of these values without transferring the ownership. These references have its own lifetime, which can be explicitly defined by the programmer or implicitly inferred by rusts compiler. This ensures that the references are valid and do not exist longer as needed. Hence this also can play a role in lock-freedom, which is needed for wait-free synchronization, since shared resources can be shared with this ownership concept. Additionally rust is a type-safe language, which can be helpfull during implementation to avoid bugs and errors, since this also needs to be avoided in a RTS. As seen rust is a good choice for implementing wait-free synchronization mechanisms for IPC in RTS. [9], [29].

Further mechanisms on how with rust different kind of common memory safety issues are solved will not be discussed in detail, as that would go beyond the scope of this thesis. It is

only important to know the basics on how rust is a type-safe and memory-safe programming language, to understand why rust is used for this work.

3. Related Work

The early foundations regarding wait-freedom was done indirectly by Leslie Lamport in 1977 and 1983 [30], [31]. While these works did not directly address or formally define wait-freedom, they laid the groundwork for lock-freedom, which Maurice Herlihy later extended to wait-freedom [1].
In 1977, Lamport he showed how one writer and multiple readers can share data without the need of locks, eliminating writer delays due to reader interference. It works by using atomic byte-wise read and write operations. The writer will atomically write a value to a memory location from right to left, while the reader uses the same memory location to read the value from left to right. So even if the writer is still in the process of writing, the reader will not block the writer while reading (and vice versa) and still see a correct snapshot of the data. To prevent a reader process from saving inconsistent data (for example while the writer process writes), the writer process will tag each update by incrementing a start counter before an end counter after writing. Readers take the start counter, read the data, and compare if the start counter matches the end counter. If they do not match it means the data is inconsistent and the reader will retry reading the data until it gets a consistent value. To understand this better imagine a date with the format DD/MM/YYYY, where every digit is written as one byte from right to left and read from left to write. This solves the problem of contention between readers and a single writer. If multiple writers are involved, the writers still have to be mutually exclusive, which means that they have to use locks to block each other to prevent inconsistency (So with multiple writers this algorithm cannot be lock-free anymore). [30]

In 1983 Leslie Lamport then gave a formal method to write and prove the correctness of any concurrent module in a simple and modular way independent of the used data structure, which he refers to as modules. These modules consist of three components:

- state functions, which are abstract variables describing the module's state.
- initial conditions, which are predicates on the state functions.
- properties, which are a mix of safety and liveness requirements.

Safety requirements define what must never happen (e.g., a queue must never drop an element), while liveness requirements define what must eventually happen (e.g., a non-empty queue must eventually allow dequeueing). He also defines the usage of action sets and environment constraints, which seperate the module action from the environments (e.g., the program where the data structure runs in). For example, a First In First Out (FIFO) queue specification would include:

- State Functions: queue contents, operation parameters, return values.
- Initial Conditions: queue starts empty.
- Safety Condition: queue maintains FIFO ordering.
- Liveness Condition: dequeue operation will eventually return a value.

This systematic methodology to prove the correctness of concurrent data structures laid the necessary groundwork for later developments too. [31]

Building on methodologys to prove concurrent data structure correctness Herlihy and Wing provided linearizability as a correctness condition for concurrent objects, which is a guarantee that every operation performed appears to take effect instantaneously at some point between the call and return of the operation [32]. This correctness condition was then used to formalize wait-freedom in 1991 [1]. In the latter work Herlihy proved that any sequential data structure can be transformed into a wait-free concurrent data structure [1]. A wait-free data structure must satisfy following three constraints:

- Linearizability: operations take effect instantaneously at some point between the call and return of the operation.
- Bounded steps: operations end in a finite number of steps.
- Independence: operations finish regardless of other processes' exection (for later understrandin: a process waiting for another process for a maximum number of time and then returning an error if that time is exceeded would still be considered wait-free, since it will finish regardless of the other process).

From this perspective the algorithm provided by Lamport in 1977 is already wait-free, even though the term was not yet defined at that time. [1], [32]

Herlihys universal construction and principles (or work that builds upon his work) appear conceptually throughout all of these wait-free algorithms [4], [27], [28], [33]–[49]. [1]

Kogan and Petrank later invented a method called fast-path slow-path, were first usually a lock-free method (the fast-path) is used to try to complete an operation, since lock-free algorithms are usually faster than wait-free algorithms. These lock-free paths are bounded by a maximum number of steps and if the operation does not complete within that bound the algorithm tries to complete the operation using a wait-free method (the slow-path). So in cases where the fast-path succeeds often without switching to the slow path, the algorithm in general completes in a shorter time then a pure wait-free algorithm. This method is used by two algorithms with one of them having a great performance advantage, which will be demonstrated later in this thesis [38], [40]. [4]

4. Methodology

To achieve the goal that is defined in section 1.2, first all wait-free data structures that could be used for IPC through shared memory in HRTS need to be found. To do this a method was used that is more known from mapping studys or literature reviews. Multiple python scripts were implemented to do this [50]. A web scraper script was written to scrape over google scholar with the following querys:

- “wait-free queue”
- “wait-free” (“mpmc” OR “multi-producer multi-consumer” OR “multi-writer multi-reader” OR “many-to-many”) “queue”
- “wait-free” (“mpsc” OR “multi-producer single-consumer” OR “single-writer multi-reader” OR “many-to-one”) “queue”
- “wait-free” (“spsc” OR “single-producer single-consumer” OR “single-writer single-reader” OR “one-to-one”) “queue”
- “wait-free” (“spmc” OR “single-producer multi-consumer” OR “multi-writer single-reader” OR “one-to-many”) “queue”

In google scholar a whitespace is considered as an AND. The rest is interpreted as read. With this approach i got a list of 1324 papers. The papers were then written into a csv file split into query, rank (number of paper), title, year, authors, venue, citations, abstract snippet, full_abstract and url with “;” as an delimiter. To extract all of this the information in google scholar was extracted and then for the full abstract the scraper went onto the source url and extracted the abstract there. If the url was a direct pdf link, a pdf reader was used to find the abstract. If an abstract was not extractable (some source site which was not considered or other problems), “ABSTRACT_NOT_FOUND” was written instead or if the paper was accessible the full paper was written instead into that cell. Because a lot of scientific web pages will put ceptchas if continuously requests are made undetected_chromedriver was importet and used as the web driver. It is an enhanced version of chromedriver which bypasses anti bot detections. After that I also implemented a regex analyzer to analyze the abstracts of the found papers on the words “lockfree”, “waitfree”, and “obstructionfree” and also again without a hyphon inbetween these words. If these keywords were not found in the abstract or the abstract contained the flag “ABSTRACT_NOT_FOUND” the paper was removed from the csv. This left 475 papers that contained at least one of these words in their abstract. After that the duplicates were removed by the algorithm, which left 325 papers. The duplicates were

removed by checking the url of the paper. Now what was left had to be manually analyzed to see if the paper was relevant for the topic. Since libre office and microsoft excel has a limit of 32.767 charachters per cell a abstract splitter had to be build to split the abstract into multiple cells. Analyzing was done by reading the paper and checking if the paper was developing an wait-free fifo queue. While dooing that also backward and forward search was done to even find more papers. Papers where only included, if and only if:

- All three of Herlihys contraints were met, that we listed in chapter 3.
- The algorithms could be implemented on x86 architecture using rust, since this is the architecture and programming languge that is used and available for this work.
- The algorithm runs in an acceptable time to even benchmark it for an IPC via a shared memory use-case.

In the end 17 papers were included from the 325 papers and 3 more paper were included by backward and forward search of the papers that were included. The papers were then split into 4 contention categories:

- MPMC queues [33], [35], [37]–[40]
- MPSC queues [28], [41], [43], [44]
- SPMC queues [27]
- SPSC queues [31], [45], [47]–[49]

Now the wait-free fifo queues had to be compared performance wise. Some papers were just improvements of other papers so only the improved version was used for the comparison. Some other papers showed multiple ways of implementing a wait-free fifo datastructure. In the end 6 MPMC queues, 4 MPSC queues, 1 SPMC queue and 11 SPSC queues were included into this work.

5. Analyzing existing Wait-Free Data Structures and Algorithms

5.1. Optimal Wait-Free Data Structure

An important question is what data structure to use for the implementation of a wait-free synchronisation technique for IPC. M. Herlihy showed that every sequential data structure can be made wait-free [1]. So it is important to choose the optimal data structure for our use. Considering that the reason of this work is to optimize modern manufacturing and automation, some form of correct data flow order is as well necessary for correct work flow for instance in an modern manufacturing line or more critical in a driverless car. Hence an already natural fit like FIFO queues. Natural because in such queues a producer process can enqueue messages and the consumer process can dequeue messages sequentially. This models real-world data flows (sensor readings, commands, network packets), which are inherently sequential. Consequently with such queues the order of the data flow is preversed without the need of implementing additional functionalities. In contrast, data structures like stacks, sets, or maps do not maintain this kind of arrival order and moreover add semantics like Last In First Out (LIFO) order or key-value pairs, which are in most cases not desired or even unnecessary. This would bring in the need of additional functions to just get rid of undesired side effects. Furthermore in a queue only two operations exist, an enqueue and an dequeue operation. All the other data structures introduce more operations and therefore more complexity and therefore more performance overhead. The less operations exist, the less complex the implementation will be. Because of these advantages and also because of the fact that in most publications in the wait-free domain queues are beeing used, limiting this thesis to queues only is reasonable. [43]

5.2. Wait-Free Algorithms

With the appropriate data structure established, an important consideration is the selection of suitable algorithms. In chapter 4 4 different contention categories are defined. Which kind of algorithm going to be used will be decided contention based. Since all of them have different complexity in runtime, it is important to choose the right contention category for the right use case to save resources and have faster execution times to meet the timing constraints of HRTS. In modern manufacturing and automation devices are used which can run multiple applications on a single device. This could mean that every application running on one device could be a producer and a consumer to each other (MPMC) and also maybe some single application of all applications running on one device produces data for just a single other

consuming application (SPSC). And maybe some single application is a producer for multiple consuming applications (SPMC) and multiple applications are producers of a single consuming application (MPSC). So it can be that all cases can occur in just one device. This means that all the different cases of contention have to be considered. In the following the different cases and their algorithms will be discussed. Moreover they will be implemented and their performance tested via a benchmarked (how fast an algorithm can produce and consume items concurrently). Subsequently from each category the best algorithm will be chosen and their performance will be compared with each other to Identify, if 4 different categories are necessary. The reason for that is, that for instance the best performed MPMC algorithm could outperform all other algorithms even for their contention category, since a MPMC approach can cover all contention cases. The goal with this approach is to have as little overhead as possible, since an algorithm explicitly implemented for a MPMC use case could have extreme overhead for a SPSC case. The implementation will be discussed in **??** and the results of the performance readings will be discussed in chapter 7. The following subsections will give an overview of the different contention categories and their algorithms found. The subsections will also shortly describe how the enqueue and dequeue in these algorithms work. Given that all algorithms found are about inter-thread communication and not IPC, the following explanation of the specific algorithms will include the terminology of threads to be precise about the papers. In **??** it will be elaborated how these thread based algorithms are adapted to IPC in Rust. Other minor Rust-specific deviations from the following algorithms (different types required by Rust's safety model, additional memory fences, etc.) can be seen in the GitHub repository accompanying this thesis [50]. These are not detailed here as such explanations would provide limited value to the topic of this work, and while a comprehensive analysis would be relevant, it would require more extensive exploration than is feasible within the scope of this work.

5.2.1. Single Producer Single Consumer (SPSC)

This is the most simple form of IPC. In SPSC there is nearly no contention from other processes, because only one producer and one consumer is working. The only contention is between the consumer and producer. This leads to the producer and consumer finish in a bounded number of steps withoput special synchronisation techniques like helping or atomic primitives. The only concern is that the data the consumer reads is consistent. Different approaches were testet in different paper, that will be seen here. Since Batched Lamport Queue (BLQ), Lazy Lamport Queue (LLQ), Batched Improved FastForward Queue (BIFFQ) and Improved FastForward Queue (IFFQ) are from the same paper, explanations and variables are shared between these algorithms to avoid redundancy:

Lamport's Circular Buffer Queue

Uses a circular array with two shared indices for synchronization, based on the algorithm originally proposed by Leslie Lamport in 1983 [31] and shown here in algorithm 2 following

Maffione et al.'s version [48]. The producer first checks if the queue is full (reached capacity N) in line 2, which requires reading the consumer's `read` index. If the queue is not full, it writes the input data to the slot at position `write & mask` in line 5, where the bitwise AND operation wraps the index around when it reaches the array end. The producer then increments `write` to signal that the written data is available in line 7. The consumer mirrors this behavior by checking if the queue is empty in line 12, which requires reading the producer's `write` index. If the queue is not empty, the consumer reads data from the slot at position `read & mask` in line 16, using the same modulo arithmetic through bitwise AND, and incrementing its `read` index to signal that the slot is available in line 17. This wraparound behavior creates the circular buffer structure, allowing the fixed-size array to be reused continuously. Unfortunately each operation requires accessing both shared indices plus the data slot, causing up to three cache misses per item when the queue moves between nearly empty and nearly full states. According to Drepper, cache misses occur when requested data is not in the local CPU core's cache and must be fetched from another core's cache, with the cache coherence protocol ensuring memory consistency across all cores [51]. Drepper showed that performance can degrade by 390%, 734%, and 1,147% for 2, 3, and 4 threads respectively. This happens because cache lines, the 64-byte blocks (on x86 architectures) that move between CPU caches, ping-pong between the producer's and consumer's cores as they take turns accessing the same memory locations [51].

Algorithm 2 Lamports Queue [48]

```
1:  function LQ_ENQUEUE(q, e)
2:      if q.write − q.read = N then                          ▷ Check if full
3:          return −1                                         ▷ No space
4:      end if
5:      q.slots[q.write ∧ q.mask] ← e
6:      store_release_barrier()
7:      q.write ← q.write + 1
8:      return 0
9:  end function
10:
11: function LQ_DEQUEUE(q)
12:     if q.read = q.write then                              ▷ Check if empty
13:         return NULL_ELEM                                  ▷ Queue empty
14:     end if
15:     load_acquire_barrier()
16:     e ← q.slots[q.read ∧ q.mask]
17:     q.read ← q.read + 1
18:     return e
19: end function
```

[48]

Lazy Lamport Queue (LLQ)

Reduces the described cache misses by postponing index reads until necessary, as shown in algorithm 3. Additionally to Lamports original enqueue function the producer maintains a local `read_shadow` copy and only updates it when running out of known free slots in lines

2 to 6. Similarly, the consumer uses `write_shadow` to avoid repeatedly checking for new items. Additionally, LLQ keeps K slots (where K is slots per cache line) permanently empty, preventing producer and consumer from touching the same cache line when the queue is full. This works well when one thread is faster than the other, reducing worst-case misses from 3 to about 2 per item. [48]

Algorithm 3 LLQ Operations [48]

```
1:  function LLQ_ENQUEUE(q, e)
2:      if q.write − q.read_shadow = N − K then                ▷ Lazy load check
3:          q.read_shadow ← q.read                             ▷ Update shadow
4:          if q.write − q.read_shadow = N − K then
5:              return −1                                      ▷ No space
6:          end if
7:      end if
8:      q.slots[q.write ∧ q.mask] ← e
9:      store_release_barrier()
10:     q.write ← q.write + 1
11:     return 0
12: end function
13:
14: function LLQ_DEQUEUE(q)
15:     if q.read = q.write_shadow then                        ▷ Lazy load check
16:         q.write_shadow ← q.write                           ▷ Update shadow
17:         if q.read = q.write_shadow then
18:             return NULL_ELEM
19:         end if
20:     end if
21:     load_acquire_barrier()
22:     e ← q.slots[q.read ∧ q.mask]
23:     q.read ← q.read + 1
24:     return e
25: end function
```

Batched Lamport Queue (BLQ)

Extends LLQ with explicit batching to further reduce synchronization costs, as detailed in algorithm 4. The producer accumulates items using private `write_priv` in line 11, filling slots without updating the shared `write` index. Only the function `blq_enqueue_publish` in lines 15 to 18 makes the batch visible by advancing `write` in line 17. The consumer works symmetrically, using `read_priv` in line 32 for local progress before updating `read` in line 40. With typical batch sizes like $B = 32$, synchronization overhead is amortized across operations, reducing cache misses. However, the application using this queue design must explicitly call the publish functions even with partial batches to avoid unbounded latency, because items remain invisible to the consumer until published. This design particularly benefits applications that naturally process data in batches, such as network packet processing, where batch boundaries are well-defined. [48]

Algorithm 4 BLQ Operations [48]

```
1: function BLQ_ENQUEUE_SPACE(q, needed)
2:     space ← N − K − (q.write_priv − q.read_shadow)
3:     if space < needed then
4:         q.read_shadow ← q.read                                  ▷ Update shadow
5:         space ← N − K − (q.write_priv − q.read_shadow)
6:     end if
7:     return space
8: end function
9:
10: function BLQ_ENQUEUE_LOCAL(q, e)
11:     q.slots[q.write_priv ∧ q.mask] ← e
12:     q.write_priv ← q.write_priv + 1
13: end function
14:
15: function BLQ_ENQUEUE_PUBLISH(q)
16:     store_release_barrier()
17:     q.write ← q.write_priv
18: end function
19:
20: function BLQ_DEQUEUE_SPACE(q)
21:     available ← q.write_shadow − q.read_priv
22:     if available = 0 then
23:         q.write_shadow ← q.write                                ▷ Update shadow
24:         available ← q.write_shadow − q.read_priv
25:     end if
26:     return available
27: end function
28:
29: function BLQ_DEQUEUE_LOCAL(q)
30:     load_acquire_barrier()
31:     e ← q.slots[q.read_priv ∧ q.mask]
32:     q.read_priv ← q.read_priv + 1
33:     return e
34: end function
35:
36: function BLQ_DEQUEUE_PUBLISH(q)
37:     q.read ← q.read_priv
38: end function
```

FastForward Queue (FFQ)

Synchronization by embedding control information directly within the data slots, eliminating separate shared indices shown in algorithm 5. Unlike Lamport's queue which requires checking both `head` and `tail` indices, FFQs producer simply examines if the next slot contains `NULL` in line 2 before writing. When the slot is empty (`NULL`), the producer writes the data directly and advances its private `head` index in lines 5 and 6. The consumer follows a similar pattern by reading from the current slot position in line 11 and then checks in line 12 data is present (non-`NULL`). If not NULL it retrieves the value and writes `NULL` to mark the slot empty in line 15. Then it advances its private `tail` index in the line after that. This couples synchronization control and the actual data, reducing shared memory accesses from three (`head`, `tail`, `buffer`) to just one (`buffer` slot). Each thread maintains its own private index that never needs synchronization. The producer tracks where to write next through `head`, while the

consumer tracks where to read next through `tail`. The `NULL` value serves dual purpose as both an empty indicator and the synchronization mechanism. While this approach reduces memory barriers and cache misses significantly, it still has the ping-pong effect when the queue has few elements, causing the producer and consumer to operate on the same cache line. [49]

Algorithm 5 FFQ Operations [49]

```
1: function FFQ_ENQUEUE(q, data)
2:     if q.buffer[q.head] ≠ NULL then
3:         return EWOULDBLOCK
4:     end if
5:     q.buffer[q.head] ← data
6:     q.head ← NEXT(q.head)
7:     return 0
8: end function
9:
10: function FFQ_DEQUEUE(q)
11:     data ← q.buffer[q.tail]
12:     if data = NULL then
13:         return EWOULDBLOCK
14:     end if
15:     q.buffer[q.tail] ← NULL
16:     q.tail ← NEXT(q.tail)
17:     return data
18: end function
```

Improved FastForward Queue (IFFQ)

Prevents cache conflicts through spatial separation using a look-ahead mechanism shown in algorithm 6. The producer checks if a slot H positions ahead (4 cache lines ahead) is empty before proceeding in line 4, ensuring it works far ahead of the consumer. This check happens only once every H items when `write` reaches `limit` in line 2. The consumer delays clearing slots through the function `iffq_dequeue_publish` seen in lines 23 to 28, maintaining separation between producer and consumer regions. With $2H$ permanently unused slots as a buffer zone, producer and consumer operate on different cache lines, reducing cache misses even more. [48]

Batched Improved FastForward Queue (BIFFQ)

Addresses IFFQs weakness when the queue is nearly empty by adding producer-side buffering, as shown in algorithm 7. Items first accumulate in a thread-local buffer seen in line 10, then the function `biffq_enqueue_publish` beginning at line 15 writes them to the queue in a rapid burst in lines 15 to 18. Also like in BLQ the application using this queue must call this function explicitly to avoid deadlocks. This behavior creates an intended race condition, which is beneficial if all writes complete before the consumer notices. The cache line stays with the producer to avoid ping-pong effects. The consumer side remains unchanged from IFFQ. While theoretical worst-case behavior is similar to IFFQ, practical measurements show

Algorithm 6 IFFQ Operations [48]

```
1: function IFFQ_ENQUEUE(q, e)
2:     if q.write = q.limit then                                  ▷ Check limit
3:         next_limit ← q.limit + H
4:         if q.slots[next_limit ∧ q.mask] ≠ NULL_ELEM then
5:             return −1                                          ▷ No space
6:         end if
7:         q.limit ← next_limit                                   ▷ Free partition
8:     end if
9:     q.slots[q.write ∧ q.mask] ← e
10:    q.write ← q.write + 1
11:    return 0
12: end function
13:
14: function IFFQ_DEQUEUE_LOCAL(q)
15:    e ← q.slots[q.read ∧ q.mask]
16:    if e = NULL_ELEM then
17:        return NULL_ELEM
18:    end if
19:    q.read ← q.read + 1
20:    return e
21: end function
22:
23: function IFFQ_DEQUEUE_PUBLISH(q)
24:    while q.clear ≠ next_clear(q.read) do
25:        q.slots[q.clear ∧ q.mask] ← NULL_ELEM
26:        q.clear ← q.clear + 1
27:    end while
28: end function
```

significant improvement when the queue operates near empty, making BIFFQ effective across all operating conditions. [48]

Algorithm 7 BIFFQ Operations [48]

```
1: function BIFFQ_WSPACE(q, needed)
2:     space ← q.limit − q.write
3:     if space < needed then
4:         return space                                           ▷ Force limit update
5:     end if
6:     return space
7: end function
8:
9: function BIFFQ_ENQUEUE_LOCAL(q, e)
10:    q.buf[q.buffered] ← e                                      ▷ Store in buffer
11:    q.buffered ← q.buffered + 1
12: end function
13:
14: function BIFFQ_ENQUEUE_PUBLISH(q)
15:    for i ← 0 to q.buffered − 1 do
16:        q.slots[q.write ∧ q.mask] ← q.buf[i]                   ▷ Fast burst
17:        q.write ← q.write + 1
18:    end for
19:    q.buffered ← 0
20:    q.limit ← q.write + H                                      ▷ Update limit
21: end function
```

B-Queue

Addresses the deadlock issues inherent in batching approaches through a self-adaptive backtracking mechanism that dynamically adjusts to production rates shown in algorithm 8. The producer maintains local `head` and `batch_head` pointers, probing `BATCH_SIZE` positions ahead when needed in line 3. The consumer's adaptive backtracking algorithm from lines 27 to 41 maintains a `batch_history` variable that records successful batch sizes from previous operations. When searching for data, it starts from this historical value rather than always beginning at `BATCH_SIZE`, significantly reducing latency when the producer operates slowly. If in line 29 to 31 the recorded size is below `BATCH_MAX`, the algorithm optimistically increments `batch_size` by `INCREMENT` (typically one cache line) to probe for higher throughput when the producer accelerates. The binary search then proceeds from this adaptive starting point, halving the batch size until finding available data or reaching zero. In the dequeue function, the consumer uses this computed value to update `batch_tail` in line 15. This eliminates the need for manual parameter adjustment or manual calling of a publish function while maintaining cache line separation and preventing deadlocks. [47]

Dyncamic Single Producer Single Consumer (dSPSC)

A dynamically space allocating queue using a linked list with node caching to reduce memory allocation overhead, as shown in algorithm 9. Unlike bounded circular buffers like the Lamport Queue, dSPSC dynamically allocates nodes as needed, making it suitable for scenarios where queue size cannot be predetermined. The implementation maintains a dummy head node `head` to ensure producer and consumer always operate on different nodes to prevent cache line conflicts. The `SPSC_Buffer` (line 4, which is just a Lamport Queue) serves as a node cache to recycle deallocated nodes to minimize malloc or free calls. When pushing, the producer first checks the cache for a recycled node in line 8, falling back to malloc only when the cache is empty in line 11. After setting the data and next pointer, a memory barrier ensures correct ordering before linking the new node into the list in lines 20 to 22. The consumer checks for available data by testing if the dummy head points to a data node in line 28. Upon a successful pop, the consumer advances the head pointer so the data node becomes the new dummy and then attempts to cache the old dummy for reuse in lines 30 to 33. While node caching improves performance, reading and referencing the pointer so often causes memory accesses spread over multiple cache lines. As shown earlier this leads to cache misses. [45]

Unbounded Single Producer Single Consumer (uSPSC)

An unbounded queue that links multiple Lamport Queues to combine the cache efficiency of Lamports circular buffer queues with unlimited capacity, as shown in algorithm 10. Unlike dSPSC which uses scattered linked list nodes, uSPSC maintains spatial locality by keeping data in contiguous circular buffers while only linking the buffers themselves. The implementation uses two pointers `buf_w` pointing to the producer's current write buffer and `buf_r` pointing to the consumer's current read buffer. When pushing, the producer checks if the current buffer

Algorithm 8 B-Queue with Self-Adaptive Backtracking [47]

```
1: function BQUEUE_ENQUEUE(q, e)
2:     if q.head = q.batch_head then                                  ▷ No empty slots
3:         if q.buffer[(q.head + BATCH_SIZE) mod q.size] ≠ NULL then
4:             return −1                                              ▷ Queue full
5:         end if
6:         q.batch_head ← q.head + BATCH_SIZE
7:     end if
8:     q.buffer[q.head mod q.size] ← e
9:     q.head ← q.head + 1
10:    return 0
11: end function
12:
13: function BQUEUE_DEQUEUE(q)
14:    if q.tail = q.batch_tail then                                  ▷ No filled slots
15:        batch_tail ← ADAPTIVE_BACKTRACK(q)
16:        if batch_tail = −1 then
17:            return NULL
18:        end if
19:        q.batch_tail ← batch_tail
20:    end if
21:    e ← q.buffer[q.tail mod q.size]
22:    q.buffer[q.tail mod q.size] ← NULL
23:    q.tail ← q.tail + 1
24:    return e
25: end function
26:
27: function ADAPTIVE_BACKTRACK(q)
28:    batch_size ← q.batch_history                                   ▷ Start from historical value
29:    if batch_size < BATCH_MAX then
30:        batch_size ← batch_size + INCREMENT                        ▷ Try larger batch
31:    end if
32:    while batch_size > 0 do
33:        batch_tail ← q.tail + batch_size
34:        if q.buffer[(batch_tail − 1) mod q.size] ≠ NULL then
35:            q.batch_history ← batch_size                           ▷ Remember successful size
36:            return batch_tail
37:        end if
38:        batch_size ← batch_size/2                                  ▷ Binary search
39:    end while
40:    return −1
41: end function
```

is full in line 2, and if so, requests a new buffer from the pool via `next_w()` in line 3 before writing the data to `buf_w`. The consumer first checks if its current buffer is empty in line 10. If empty, it determines whether the queue is truly empty by comparing read and write buffer pointers in line 11. If they point to the same buffer, no more data exists. Otherwise, after rechecking emptiness to prevent race conditions in line 14, the consumer obtains the next buffer via `next_r()` and releases the empty buffer back to the pool for recycling in lines 15 to 17. This double-check prevents data loss when the producer writes to the current buffer between the initial emptiness check and the buffer comparison. By reusing entire buffers rather than individual nodes, uSPSC matches bounded SPSC queues cache behavior while providing unbounded capacity. [45]

Algorithm 9 dSPSC Operations [45]

```
1: struct Node { void* data; Node* next; }
2: Node* head;                                          ▷ Points to dummy node
3: Node* tail;                                          ▷ Points to last data node
4: SPSC_Buffer cache;                                   ▷ Bounded cache for node recycling
5:
6: function ALLOCNODE
7:     Node* n ← NULL
8:     if cache.pop(&n) then                            ▷ Try cache first
9:         return n
10:    end if
11:    n ← (Node*)malloc(sizeof(Node))
12:    return n
13: end function
14:
15: function PUSH(void* data)
16:    Node* n ← allocnode()                            ▷ Get node from cache or malloc
17:    n->data ← data
18:    n->next ← NULL
19:    WMB()                                            ▷ Write Memory Barrier
20:    tail->next ← n                                   ▷ Link new node
21:    tail ← n                                         ▷ Update tail pointer
22:    return true
23: end function
24:
25: function POP(void** data)
26:    if head->next ≠ NULL then                        ▷ Check if data available
27:        Node* n ← head                               ▷ Save current dummy
28:        *data ← (head->next)->data                   ▷ Extract data
29:        head ← head->next                            ▷ Advance to next node
30:        if !cache.push(n) then                       ▷ Try to recycle old dummy
31:            free(n)                                  ▷ Free if cache full
32:        end if
33:        return true
34:    end if
35:    return false                                     ▷ Queue empty
36: end function
```

MultiPush Single Producer Single Consumer (mSPSC)

Reduces the ping-pong effect in Lamport's circular buffer by batching multiple elements before insertion, as shown in algorithm 11. Instead of writing elements one by one directly to the shared buffer, mSPSC accumulates items in a thread-local array `batch`. The producer stores incoming data in the batch array in lines 2 and 3, and when the batch reaches `BATCH_SIZE` in line 4, the producer calls `multipush` to insert all elements at once in line 5. The `multipush` function first calculates the final write position in line 11 and checks if sufficient space exists in line 12. As seen in lines 15 to 17 elements are written in reverse order, starting from the furthest position and working backwards. This backward insertion creates distance between the write pointer and where the consumer is reading, ensuring they operate on different cache lines. A write memory barrier in line 18 ensures all batch writes are visible before updating the write pointer in line 19. The batch counter resets in line 20, preparing for the next batch. The `flush` function in lines 24 to 29 allows forcing partial batch writes when needed. While

Algorithm 10 uSPSC Operations[45]

```
1: function USPSC_PUSH(q, data)
2:     if q.buf_w.full() then                          ▷ Current buffer full
3:         q.buf_w ← q.pool.next_w()                   ▷ Get new buffer
4:     end if
5:     q.buf_w.push(data)
6:     return true
7: end function
8:
9: function USPSC_POP(q, data)
10:     if q.buf_r.empty() then
11:         if q.buf_r = q.buf_w then                  ▷ Same buffer?
12:             return false                           ▷ Queue truly empty
13:         end if
14:         if q.buf_r.empty() then                    ▷ Recheck after comparison
15:             tmp ← q.pool.next_r()
16:             q.pool.release(q.buf_r)                ▷ Recycle buffer
17:             q.buf_r ← tmp
18:         end if
19:     end if
20:     return q.buf_r.pop(data)
21: end function
```

adding an extra copy per element from batch to buffer, the improved cache behavior from reduced traffic from the coherence protocol compensates for this overhead. [45]

Algorithm 11 mSPSC Operations[45]

```
1: function MSPSC_PUSH(q, data)
2:     q.batch[q.count] ← data
3:     q.count ← q.count + 1
4:     if q.count = BATCH_SIZE then
5:         return MULTIPUSH(q, q.batch, q.count)
6:     end if
7:     return true
8: end function
9:
10: function MULTIPUSH(q, batch, len)
11:     last ← q.write + len − 1                       ▷ Calculate end position
12:     if q.slots[last mod q.size] ≠ NULL then
13:         return false                               ▷ Not enough space
14:     end if
15:     for i ← len − 1 downto 0 do                    ▷ Reverse order
16:         q.slots[(q.write + i) mod q.size] ← batch[i]
17:     end for
18:     WMB()                                          ▷ Ensure all writes visible
19:     q.write ← (last + 1) mod q.size
20:     q.count ← 0                                    ▷ Reset batch counter
21:     return true
22: end function
23:
24: function FLUSH(q)
25:     if q.count > 0 then
26:         return MULTIPUSH(q, q.batch, q.count)
27:     end if
28:     return true
29: end function
```

Jayanti SPSC Queue

A queue specifically for composability in larger MPSC structures, as shown in algorithm 12. Unlike traditional SPSC queues, this implementation includes `readFront` operations that enables observation of the queue's head element, crucial for the MPSC construction. The queue maintains a linked list where the tail always points to a dummy node. When enqueueing, the producer converts the current dummy node into a data node by writing the value in line 4. Then the producer links a new dummy node in line 5 and updates the tail pointer in line 6. This ensures the consumer never sees a partially constructed node. The `Help` variable in line 15 stores the dequeued value, allowing concurrent `readFront_e` operations to obtain valid data even after the original node is removed. The announcement mechanism prevents use-after-free errors. When the producer calls `readFront_e`, it writes the front node pointer to `Announce` in line 32, signaling the consumer not to immediately free that node. If the consumer encounters an announced node in line 17, it defers the node's deallocation to `FreeLater` in lines 18 to 20, ensuring the producer can safely read the node's value. The separate `readFront_d` operation in lines 41 to 47 is simpler since the consumer knows no concurrent dequeue can occur. This coordination enables wait-free progress while supporting the propagation mechanism, the process of pushing each local queue's minimum timestamp up through a binary tree to maintain a global minimum, needed for the logarithmic-time MPSC operations as seen in section 5.2.2. [44]

5.2.2. Multi Producer Single Consumer (MPSC)

This is a bit more complex to implement than the SPSC case. Multiple producers can enqueue items at the same time while a single consumer dequeues items. This includes implementing other strategies then just adding some memory barriers, such as helping and atomic primitives to maintain wait-freedom and consistency between the producers. There are multiple approaches that are available from different papers to achieve this:

Jayanti MPSC Queue

Achieves logarithmic time complexity by distributing the global queue across n local SPSC queues implemented like in section 5.2.1 and organized under a binary tree, as shown in algorithm 13. Each producer owns a dedicated local queue, eliminating producer-producer contention. When enqueueing, a producer obtains a global timestamp via LL/SC on a shared counter in lines 2 and 3, creating a unique ordering even if the SC fails, since some other producer must have incremented it. If LL/SC is not supported on system architecture it can be replaced with versioned CAS. The producer then inserts a timestamped pair into its local queue in line 4 and propagates this timestamp up the tree in line 5. The tree maintains the invariant that each internal node holds the minimum timestamp of its subtree. The `propagate` function in lines 18 to 26 walks from leaf to root, calling `refresh` at each node. The double `refresh` pattern in lines 22 to 24 ensures correctness. if the first refresh fails, another process updated the node. if the second refresh also fails, that process must have read the updated

Algorithm 12 Jayanti's SPSC Queue Operations [44]

```
1: function ENQUEUE(q, v)
2:     newNode ← new Node()                          ▷ Create new dummy node
3:     tmp ← q.Last                                  ▷ Get current dummy tail
4:     tmp.val ← v                                   ▷ Convert dummy to data node
5:     tmp.next ← newNode                            ▷ Link new dummy
6:     q.Last ← newNode                              ▷ Update tail pointer
7: end function
8:
9: function DEQUEUE(q)
10:    tmp ← q.First                                 ▷ Get head node
11:    if tmp = q.Last then                          ▷ Only dummy remains?
12:        return ⊥                                  ▷ Queue empty
13:    end if
14:    retval ← tmp.val                              ▷ Read value
15:    q.Help ← retval                               ▷ Help concurrent readFront
16:    q.First ← tmp.next                            ▷ Remove from queue
17:    if tmp = q.Announce then                      ▷ Was announced by readFront?
18:        tmp′ ← q.FreeLater                        ▷ Get old deferred node
19:        q.FreeLater ← tmp                         ▷ Defer current node
20:        free(tmp′)                                ▷ Free old deferred node
21:    else
22:        free(tmp)                                 ▷ Free immediately
23:    end if
24:    return retval
25: end function
26:
27: function READFRONT_E(q)                          ▷ Called by enqueuer
28:    tmp ← q.First                                 ▷ Read head pointer
29:    if tmp = q.Last then                          ▷ Queue empty?
30:        return ⊥
31:    end if
32:    q.Announce ← tmp                              ▷ Announce to prevent free
33:    if tmp ≠ q.First then                         ▷ Head changed (was dequeued)?
34:        retval ← q.Help                           ▷ Use helped value
35:    else
36:        retval ← tmp.val                          ▷ Read directly
37:    end if
38:    return retval
39: end function
40:
41: function READFRONT_D(q)                          ▷ Called by dequeuer
42:    tmp ← q.First
43:    if tmp = q.Last then
44:        return ⊥
45:    end if
46:    return tmp.val                                ▷ Safe - no concurrent dequeue
47: end function
```

children values and installed the correct minimum. The `refresh` function uses LL/SC in lines 28 to 33 to atomically update a node with the minimum of its children's timestamps. The consumer reads the root to find the producer with the earliest element in line 9 and then dequeues from that local queue in line 13 and propagates any changes in line 14. This design transforms the $O(n)$ scan of all queues into $O(\log n)$ tree traversals, while the space complexity remains $O(n+m)$ where m is the number of queued items. [44]

Algorithm 13 Jayanti's MPSC Queue Operations [44]

```
1: function ENQUEUE(q, p, v)
2:     tok ← LL(q.counter)                          ▷ Read timestamp
3:     SC(q.counter, tok + 1)                       ▷ Try increment
4:     enqueue2(q.Q[p], (v, (tok, p)))              ▷ Add timestamp to local queue
5:     PROPAGATE(q, q.Q[p])
6: end function
7:
8: function DEQUEUE(q, p)
9:     [t, id] ← read(q.T.root)                     ▷ Get min producer
10:    if id = ⊥ then
11:        return ⊥
12:    end if
13:    ret ← dequeue2(q.Q[id])
14:    PROPAGATE(q, q.Q[id])
15:    return ret.val
16: end function
17:
18: function PROPAGATE(q, localQueue)
19:    currentNode ← localQueue
20:    repeat
21:        currentNode ← parent(currentNode)
22:        if ¬REFRESH(q, currentNode) then          ▷ First try
23:            REFRESH(q, currentNode)               ▷ Second ensures correctness
24:        end if
25:    until currentNode = q.T.root
26: end function
27:
28: function REFRESH(q, node)
29:    LL(node)                                     ▷ Load-link node
30:    stamps ← read timestamps from node's children
31:    minT ← minimum timestamp from stamps
32:    return SC(node, minT)                        ▷ Store-conditional
33: end function
```

Drescher Queue

Uses a linked list with a dummy head node to eliminate producer contention, as shown in algorithm 14. Unlike traditional MPSC queues that require retry loops, producers complete enqueue in exactly three steps. First the producers clear the item's `next` pointer in line 6, then it atomically swaps the tail pointer via `FAS` in line 7 and further links the previous tail to the new item in line 8. The `FAS` operation ensures multiple producers can enqueue concurrently without interference. The consumer reads the head and its `next` pointer in lines 12 and 13 and afterwards advances the head in line 17 if the queue is non-empty. Subsequently the consumer handles the special case of the dummy node in lines 18 to 24. When the dummy is dequeued, it's immediately re-enqueued in line 19 to maintain the invariant that the queue always contains at least one element to prevent complex empty queue conditions. The guard integration through `VOUCH` and `CLEAR` ensures all orders are enqueued before guard acquisition in line 2 of `VOUCH`. The guard is released before checking for pending orders in line 11 of `CLEAR`, preventing lost updates. This ordering guarantees that either the current

sequencer or a new thread will process all pending orders, achieves the wait-free progress guarantees with constant time enqueue operations. [28]

Algorithm 14 Drescher's Wait-Free MPSC Queue Operations

```
1:  dummy.next ← 0
2:  head ← &dummy
3:  tail ← &dummy
4:
5:  procedure ENQUEUE(guard, item)
6:      item.next ← 0                                  ▷ Clear next pointer
7:      prev ← FAS(guard.tail, item)                   ▷ Atomic swap tail
8:      prev.next ← item                               ▷ Link to new item
9:  end procedure
10:
11: function DEQUEUE(guard)
12:     item ← guard.head
13:     next ← guard.head.next
14:     if next = 0 then                               ▷ Empty queue?
15:         return ⊥
16:     end if
17:     guard.head ← next
18:     if item = &dummy then                          ▷ Dequeued dummy?
19:         ENQUEUE(guard, item)                       ▷ Re-enqueue dummy
20:         if guard.head.next = 0 then                ▷ Still empty?
21:             return ⊥
22:         end if
23:         guard.head ← guard.head.next
24:         return next
25:     end if
26:     return item
27: end function
```

Jiffy Queue

A queue that uses a linked list of fixed-size arrays (buffers), as shown in function `ENQUEUE(data)` in algorithm 15 and in `DEQUEUE` function in algorithm 16. Unlike other linked-list queues that allocate nodes per element, Jiffy amortizes allocation overhead by storing multiple elements in each buffer. Producers use FAA on a global tail counter to reserve slots in line 2 of `enqueue`, eliminating producer-producer synchronization except during buffer allocation. Each buffer contains an array of nodes with data and a 2-bit `isSet` flag indicating the node's state: `empty` (uninitialized), `set` (data written), or `handled` (already dequeued). When the current buffer fills, producers allocate new buffers and link them via CAS in lines 7 and 8. To reduce allocation contention, the producer obtaining the second slot in each buffer proactively allocates the next buffer in lines 21 to 26, ensuring smooth transitions between buffers. The consumer maintains a local head pointer and scans for the first non-handled element in lines 3 to 9 of `dequeue`. To ensure linearizability when producers stall: if the head element is still `empty`, the consumer scans forward to find a `set` element in line 20, then rescans backward in line 24 to ensure no earlier element became `set` during the scan. This prevents violating FIFO ordering when a slow producer completes after a faster

one. The consumer can "fold" the queue by deleting fully-handled buffers in the middle of the list during scans, to not use to much memory even with stalled producers. This achieves wait-free progress guarantees with minimal synchronization. Producers only need one FAA per enqueue, while the consumer performs no atomic operations at all. [43]

Algorithm 15 Jiffy MPSC Queue Enqueue Operation [43]

```
1: function ENQUEUE(data)
2:     location ← FAA(tail, 1)                          ▷ Reserve global index
3:     tempTail ← tailOfQueue
4:     while location is in unallocated buffer do       ▷ Beyond last buffer?
5:         if tempTail.next = NULL then
6:             newArr ← new BufferList()
7:             if CAS(tempTail.next, NULL, newArr) then
8:                 CAS(tailOfQueue, tempTail, newArr)
9:             else
10:                delete newArr                        ▷ Another thread succeeded
11:            end if
12:        end if
13:        tempTail ← tailOfQueue                       ▷ Move to new buffer
14:    end while
15:    while location not in tempTail's buffer do       ▷ Location in earlier buffer?
16:        tempTail ← tempTail.prev                     ▷ Walk backward
17:    end while
18:    index ← location − tempTail.startIndex           ▷ Buffer-local index
19:    tempTail.buffer[index].data ← data
20:    tempTail.buffer[index].isSet ← SET               ▷ Mark as ready
21:    if index = 1 AND tempTail is last buffer then    ▷ Second slot?
22:        newArr ← new BufferList()                    ▷ Proactive allocation
23:        if NOT CAS(tempTail.next, NULL, newArr) then
24:            delete newArr
25:        end if
26:    end if
27: end function
```

DQueue

Combines local buffering with a segmented shared queue to minimize synchronization overhead, as shown in function `ENQUEUE(data)` in algorithm 17 and in `DEQUEUE` function in algorithm 18. DQueue reduces contention by having producers accumulate enqueue requests in thread-local ring buffers before writing them to the shared queue in batches. Producers reserve slots using FAA on a global tail counter in line 7 of `enqueue`, storing both the data and the reserved cell index (`cid`) in their local buffer. Each producer maintains a buffer of `Request` structures with capacity L, using `local_head` and `local_tail` pointers to track buffer state. When the buffer fills, detected in line 2, the producer calls `dump_local_buffer` to flush all buffered requests. During flushing, producers write values directly to their reserved cells in line 15 without synchronization, as each cell is exclusively owned by the reserving producer. Producers cache their current segment pointer (`pseg`) and update it when moving to newer segments in line 18. The `find_segment` function traverses the segment list and allocates new segments on-demand using CAS in line 29. To maintain wait-freedom when

Algorithm 16 Jiffy MPSC Queue Dequeue Operation [43]

```
1: function DEQUEUE
2:     n ← headOfQueue.buffer[head]
3:     while n.isSet = HANDLED do                          ▷ Skip dequeued items
4:         head ← head + 1
5:         if end of buffer then
6:             move to next buffer and delete current
7:         end if
8:         n ← headOfQueue.buffer[head]
9:     end while
10:    if queue is empty then
11:        return ⊥
12:    end if
13:    if n.isSet = SET then                               ▷ Ready to dequeue?
14:        data ← n.data
15:        n.isSet ← HANDLED
16:        head ← head + 1
17:        return data
18:    end if
19:    if n.isSet = EMPTY then                             ▷ Incomplete enqueue?
20:        tempN ← Scan(find first SET element)
21:        if no SET element found then
22:            return ⊥
23:        end if
24:        Rescan(n, tempN)                                ▷ Check for newly set elements
25:        data ← tempN.data
26:        tempN.isSet ← HANDLED
27:        return data
28:    end if
29: end function
30:
31: function SCAN                                          ▷ Find first SET element
32:    for each element from current position do
33:        if element.isSet = SET then
34:            return element
35:        end if
36:        if entire buffer is HANDLED then
37:            fold queue (delete buffer)
38:        end if
39:    end for
40:    return NULL
41: end function
42:
43: function RESCAN(start, end)                            ▷ Check for ordering violations
44:    for each element from start to end do
45:        if element.isSet = SET then
46:            end ← element                               ▷ Found earlier SET element
47:            restart scan from start
48:        end if
49:    end for
50: end function
```

producers stall, the consumer encountering an empty cell that should contain data, checked in line 7 of `dequeue`, distinguishes between an empty queue in line 8 and a pending enqueue by checking if head equals tail. For pending enqueues, `help_enqueue` in line 11 iterates through all producers' local buffers from lines 19 to 31, writing any buffered values to their

reserved cells in line 28. The helper skips producers that have already moved past the target segment in line 24, avoiding unnecessary work. This ensures minimal synchronization with only one FAA per enqueue and no atomics for dequeue and improves cache locality via batched writes that reduce false sharing and writes that directly write to known cell locations without searching. The consumer's dequeue operation has its linearization point at line 14 where it increments the head pointer. [41]

Algorithm 17 DQueue MPSC Queue Enqueue Operation [41]

```
1: function ENQUEUE(Producer p, data)
2:     if next(p.local_tail) = p.local_head then
3:         dump_local_buffer(p)                                  ▷ Flush when full
4:     end if
5:     tail ← p.local_tail
6:     p.local_buffer[tail].val ← data
7:     p.local_buffer[tail].cid ← FAA(q.tail, 1)                 ▷ Reserve slot
8:     p.local_tail ← next(p.local_tail)
9: end function
10:
11: function DUMP_LOCAL_BUFFER(Producer p)
12:     while p.local_head ≠ p.local_tail do
13:         r ← p.local_buffer[p.local_head]
14:         seg ← find_segment(p.pseg, r.cid)
15:         seg.cell[r.cid  mod N] ← r.val                        ▷ Write in batch
16:         p.local_head ← next(p.local_head)
17:         if p.pseg ≠ seg then
18:             p.pseg ← seg                                      ▷ Update segment cache
19:         end if
20:     end while
21: end function
22:
23: function FIND_SEGMENT(Segment *sp, int cid)
24:     curr ← sp
25:     for i ← curr → id; i < cid/N; i++ do
26:         next ← curr → next
27:         if next = NULL then
28:             new ← new_segment(i + 1)
29:             if CAS(curr → next, NULL, new) then
30:                 next ← new
31:             else
32:                 delete new                                    ▷ Another thread succeeded
33:             end if
34:         end if
35:         curr ← next
36:     end for
37:     return curr
38: end function
```

5.2.3. Single Producer Multi Consumer (SPMC)

This case is trickier to implement, since now multiple reading workers have to be synchronized to read consistently without any unwanted behavior. Multiple producers was simpler, since making producers write specific data each is not so hard. In this case every consumer has to be synchronized so that when one item is consumed by a consumer, it cannot be consumed

Algorithm 18 DQueue MPSC Queue Dequeue Operation [41]

```
1: function DEQUEUE(Consumer c)
2:     seg ← find_segment(c.cseg, q.head)
3:     if c.cseg ≠ seg then
4:         c.cseg ← seg                                   ▷ Update segment cache
5:     end if
6:     cell ← seg.cell[q.head mod N]
7:     if cell = ⊥ then                                   ▷ Empty cell?
8:         if q.head = q.tail then
9:             return EMPTY
10:        else
11:            help_enqueue()                             ▷ Help stalled producers
12:        end if
13:    end if
14:    q.head ← q.head + 1                                ▷ Linearization point
15:    return cell
16: end function
17:
18: function HELP_ENQUEUE
19:    for each Producer p in system do
20:        for each Request r in p.local_buffer do
21:            pos ← r.cid
22:            val ← r.val
23:            seg ← find_segment(p.pseg, pos)
24:            if seg.id > pos/N then
25:                break                                  ▷ Producer moved past
26:            end if
27:            if seg.cell[pos mod N] = ⊥ then
28:                seg.cell[pos mod N] ← val              ▷ Help write
29:            end if
30:        end for
31:    end for
32: end function
```

again from another and also the ordering has to be kept. That is most probably also the reason only one algorithm for this contention category was found. The approach to achieve this in a wait-free manner is the following:

David Queue

Uses a two-dimensional array of Swap objects to handle the race condition where consumers overtake the producer, as shown in algorithm 19. David's queue allows the producer to detect when it has been overtaken and adapt by jumping to a fresh row. The prodcer maintains two persistent local variables, a `enq_row` (current row) and `tail` (next column to write) variable. During enqueue, the producer swaps the value into `ITEMS[enq_row, tail]` in line 10 and checks if the retrieved value is $\top$, indicating a consumer already accessed this cell. If so, the producer jumps to the next row in lines 12 to 15, writing the value to the new row and updating the shared `ROW` register. This jump mechanism ensures that enqueued values are never lost. Consumers read the active row from `ROW` in line 22 of `dequeue`, then increment using Fetch and Increment (Like FAA, but incrementing) on `HEAD[deq_row]` in line 23 to reserve a unique column index. They swap $\top$ into the reserved cell in line 24, retrieving either

the enqueued value or $\perp$ (empty). The use of swap instead of plain registers is important, because it allows the producer to detect consumer interference (by finding $\top$) and consumers to mark cells as processed. Each cell in `ITEMS` is accessed at most once by an enqueue and once by a dequeue operation. The algorithm achieves 3-bounded wait-freedom with constant time operations. The producer completes in at most 3 steps (regular enqueue: 1 step, jump enqueue: 3 steps), while consumers always complete in exactly 3 steps. [27]

Algorithm 19 David's Queue Operations [27]

```
1: Shared variables:
2: HEAD: array of Fetch&Increment objects, initially 0
3: ITEMS: 2D array of Swap objects, initially ⊥
4: ROW: Register, initially 0
5:
6: Enqueuer's persistent local variables:
7: enq_row ← 0, tail ← 0
8:
9: procedure ENQUEUE(x)                                   ▷ For enqueuer E only
10:     val ← Swap(ITEMS[enq_row, tail], x)               ▷ Try to enqueue
11:     if val = ⊤ then                                   ▷ Dequeuer overtook us?
12:         enq_row ← enq_row + 1                         ▷ Jump to next row
13:         tail ← 0
14:         Swap(ITEMS[enq_row, tail], x)                 ▷ Write to new row
15:         Write(ROW, enq_row)                           ▷ Publish new row
16:     end if
17:     tail ← tail + 1
18:     return OK
19: end procedure
20:
21: function DEQUEUE                                      ▷ For dequeuers D_1, ..., D_n
22:     deq_row ← Read(ROW)                               ▷ Get active row
23:     head ← Fetch&Increment(HEAD[deq_row])             ▷ Reserve column
24:     val ← Swap(ITEMS[deq_row, head], ⊤)               ▷ Get value
25:     if val = ⊥ then                                   ▷ Empty cell?
26:         return ε                                      ▷ Queue was empty
27:     else
28:         return val                                    ▷ Return dequeued value
29:     end if
30: end function
```

5.2.4. Multi Producer Multi Consumer (MPMC)

Finally a look into the MPMC case can be made. Here we need to think about synchronize producer-producer contention, consumer-consumer contention and producer consumer contention. This includes helping methods for the producer and consumer and different atomic primitives. Multiple approaches are available to achieve this:

Kogan and Petranks queue

Uses a priority-based helping scheme with Michael and Scott's lock-free queue from algorithm 1 as the foundation to achieve wait-freedom, as shown in algorithms 20 to 22. Threads

complete operations in bounded steps by helping slower peers. Each thread chooses a monotonically increasing phase number in line 2 of ENQUEUE in algorithm 20 and line 2 of DEQUEUE of algorithm 21, then records operation details in a shared state array in line 3. The helping mechanism in HELP from lines 15 to 26 in algorithm 22 ensures all operations with phases ≤ the current phase complete. Threads traverse the state array and invoke HELP_ENQ or HELP_DEQ based on pending operations in lines 19 to 23 of algorithm 22. For enqueue, threads utilize the tail update like in Michael and Scott by first appending the node via CAS in line 15 of HELP_ENQ in algorithm 20, then finally helping by updating the tail in line 36 of HELP_FINISH_ENQ. The three-step scheme ensures exactly-once execution by first appending a node to list in line 15 then clear the pending flag in line 35 of HELP_FINISH_ENQ, and finally update the tail pointer in line 36. For dequeue, threads write their ID to the deqTid field of the head node in line 41 of HELP_DEQ in algorithm 21 to "lock" it logically. The consumer then updates the pending flag in line 9 of HELP_FINISH_DEQ in algorithm 22 and advances head in line 10. Special handling for empty queues occurs in lines 20 to 25 of HELP_DEQ of algorithm 21, where threads update state with null to indicate emptiness. The phase selection using MAXPHASE in lines 41 to 50 in algorithm 20 ensures threads help all concurrent operations before returning, preventing starvation. This achieves wait-free progress with $O(n)$ steps per operation where n is the number of threads. [33]

Turn Queue

Uses a novel turn-based consensus mechanism to achieve wait-freedom without requiring FAA instructions. As shown in algorithms 23 to 25, the queue maintains two arrays: enqueuers for enqueue requests and deqself/deqhelp for dequeue operations. Each thread has a unique index used as its thread ID (enqTid or deqTid). For the producers in algorithm 23, threads publish their intent by storing a node pointer in enqueuers[myIdx] in line 6 of the function ENQUEUE. The consensus mechanism uses the enqTid of the tail node to determine whose turn is next. Threads scan the enqueuers array starting from position (tail->enqTid + 1) % maxThreads in line 20, helping the first non-null request they find. This creates a circular turn order ensuring fairness. The algorithm guarantees that after publishing a request, at most maxThreads-1 other nodes will be enqueued first, achieving wait-free bounded progress. The enqueue operation protects the tail pointer using hazard pointers in line 12 of the ENQUEUE function, then clears any completed request from the tail's position in lines 15 to 18. It searches for the next request to help in lines 19 to 26, attempting to append the found node via CAS in line 23. Finally, it advances the tail pointer if a node was successfully appended in line 29. The operation completes when the thread's own request has been processed (detected by checking if enqueuers[myIdx] is null in line 8). For conmsumers in algorithm 24, the algorithm uses a dual-array approach with deqself and deqhelp to avoid excessive hazard pointer usage. Threads open a request by making deqself[myIdx] equal to deqhelp[myIdx] in lines 4 and 5 of the DEQUEUE function. The turn order follows the deqTid of the head node. When assigning nodes, threads use CAS to set the next node's deqTid field in line 8 of SEARCHNEXT in algorithm 25. This assignment

Algorithm 20 Kogan and Petrank's Queue Enqueue Operation [33]

```
1: function ENQUEUE(value)
2:     phase ← MAXPHASE +1                                   ▷ Choose phase
3:     state[tid] ← OpDesc(phase, true, true, Node(value, tid))
4:     HELP(phase)                                           ▷ Help all ops ≤ phase
5:     HELP_FINISH_ENQ                                       ▷ Ensure tail updated
6: end function
7:
8: procedure HELP_ENQ(tid, phase)
9:     while ISSTILLPENDING(tid, phase) do
10:        last ← tail
11:        next ← last.next
12:        if last = tail then                               ▷ Validate read
13:            if next = null then                           ▷ Can append?
14:                if ISSTILLPENDING(tid, phase) then
15:                    if CAS(last.next, null, state[tid].node) then
16:                        HELP_FINISH_ENQ
17:                        return
18:                    end if
19:                end if
20:            else                                          ▷ Help pending enqueue
21:                HELP_FINISH_ENQ
22:            end if
23:        end if
24:    end while
25: end procedure
26:
27: procedure HELP_FINISH_ENQ
28:    last ← tail
29:    next ← last.next
30:    if next ≠ null then
31:        tid ← next.enqTid                                 ▷ Thread that owns node
32:        desc ← state[tid]
33:        if last = tail and state[tid].node = next then
34:            newDesc ← OpDesc(state[tid].phase, false, true, next)
35:            CAS(state[tid], desc, newDesc)                ▷ Clear pending
36:            CAS(tail, last, next)                         ▷ Update tail
37:        end if
38:    end if
39: end procedure
40:
41: function MAXPHASE
42:    max ← −1
43:    for i ← 0 to NUM_THREADS − 1 do
44:        phase ← state[i].phase
45:        if phase > max then
46:            max ← phase
47:        end if
48:    end for
49:    return max
50: end function
```

is permanent and indicates ownership. The dequeue handles empty queues through a "give-up" mechanism implemented in `GIVEUP` (lines 28 to 44 of algorithm 25). When detecting an empty queue (head equals tail) in line 12 of `DEQUEUE` in algorithm 24, threads rollback their request in line 13 but must ensure no concurrent thread assigned them a node. This

Algorithm 21 Kogan and Petrank's Queue Dequeue Operation [33]

```
1: function DEQUEUE
2:     phase ← MAXPHASE +1
3:     state[tid] ← OpDesc(phase, true, false, null)
4:     HELP(phase)
5:     HELP_FINISH_DEQ
6:     node ← state[tid].node
7:     if node = null then
8:         throw EmptyException
9:     end if
10:    return node.next.value
11: end function
12:
13: procedure HELP_DEQ(tid, phase)
14:    while ISSTILLPENDING(tid, phase) do
15:        first ← head
16:        last ← tail
17:        next ← first.next
18:        if first = head then                                   ▷ Validate read
19:            if first = last then                               ▷ Queue might be empty
20:                if next = null then                            ▷ Confirmed empty
21:                    desc ← state[tid]
22:                    if last = tail and ISSTILLPENDING(tid, phase) then
23:                        newDesc ← OpDesc(state[tid].phase, false, false, null)
24:                        CAS(state[tid], desc, newDesc)
25:                    end if
26:                else
27:                    HELP_FINISH_ENQ                            ▷ Help enqueue
28:                end if
29:            else                                               ▷ Queue not empty
30:                desc ← state[tid]
31:                node ← desc.node
32:                if not ISSTILLPENDING(tid, phase) then
33:                    break
34:                end if
35:                if first = head and node ≠ first then
36:                    newDesc ← OpDesc(state[tid].phase, true, false, first)
37:                    if not CAS(state[tid], desc, newDesc) then
38:                        continue
39:                    end if
40:                end if
41:                CAS(first.deqTid, −1, tid)                     ▷ Lock node
42:                HELP_FINISH_DEQ
43:            end if
44:        end if
45:    end while
46: end procedure
```

involves re-checking the queue state and potentially self-assigning the first node if no other requests exist (line 41 of `GIVEUP`). The algorithm closes requests by updating `deqhelp[i]` in line 18 of `CASDEQANDHEAD` in algorithm 25, making it differ from `deqself[i]`. Memory reclamation uses wait-free bounded hazard pointers integrated into the algorithm. Nodes are retired in line 35 of `DEQUEUE` in algorithm 24 only after ensuring they're no longer accessible through shared variables. The algorithm requires only one allocation per enqueued item (the

Algorithm 22 Kogan and Petrank's Queue Dequeue Helping Operations [33]

```
1: procedure HELP_FINISH_DEQ
2:     first ← head
3:     next ← first.next
4:     tid ← first.deqTid                                   ▷ Thread that locked node
5:     if tid ≠ −1 then
6:         desc ← state[tid]
7:         if first = head and next ≠ null then
8:             newDesc ← OpDesc(state[tid].phase, false, false, state[tid].node)
9:             CAS(state[tid], desc, newDesc)               ▷ Clear pending
10:            CAS(head, first, next)                       ▷ Advance head
11:        end if
12:    end if
13: end procedure
14:
15: procedure HELP(phase)
16:    for i ← 0 to NUM_THREADS − 1 do
17:        desc ← state[i]
18:        if desc.pending and desc.phase ≤ phase then
19:            if desc.enqueue then
20:                HELP_ENQ(i, phase)
21:            else
22:                HELP_DEQ(i, phase)
23:            end if
24:        end if
25:    end for
26: end procedure
```

node itself), achieving minimal memory overhead of $O(N_{threads})$ compared to the $O(N^2_{threads})$ of other wait-free queues. [37]

Yang Mellor-Crummey (YMC) queue

Uses FAA and CAS combined with Kogan's and Petrank's fast-path slow-path methodology mentioned in chapter 3 to use the advantage of lock-free algorithms if possible while still maintaining wait-freedom. As shown in algorithms 26 to 29, the queue represents cells as an infinite array emulated through linked segments. Each segment contains `N` cells and a pointer to the next segment. The queue maintains global indices `H` (head) and `T` (tail) that are accessed using FAA, ensuring atomic increments without retry loops. For producers in algorithm 26, threads obtain a unique cell index via FAA on `T` in line 11. They then locate the corresponding cell using `find_cell` which traverses segments and allocates new ones if needed. The fast-path attempts a simple CAS to deposit the value in line 13. If this fails, due to concurrent dequeue marking the cell unusable, the thread switches to the slow-path starting at line 20. The slow-path employs a helping mechanism where threads publish enqueue requests in their handle structure in line 23. In algorithm 29 consumers help pending enqueues through `help_enq` in line 13 when they mark cells unusable. This creates a symbiotic relationship: producers get help depositing values, while consumers ensure values are available to dequeue. For consumers in algorithm 28, threads obtain cell indices via FAA on `H` in line 18. The algorithm uses `help_enq` to secure values, which may involve helping slow-path enqueues. If a value

Algorithm 23 Turn Queue Enqueue Operation [37]

```
1: function ENQUEUE(item)
2:     if item = null then throw InvalidArgument
3:     end if
4:     myIdx ← GETINDEX
5:     myNode ← new Node(item, myIdx)
6:     enqueuers[myIdx] ← myNode
7:     for i ← 0 to maxThreads − 1 do
8:         if enqueuers[myIdx] = null then                    ▷ Request completed
9:             hp.CLEAR
10:            return
11:        end if
12:        ltail ← hp.PROTECTPTR(kHpTail, tail)
13:        if ltail ≠ tail then continue
14:        end if
15:        if enqueuers[ltail.enqTid] = ltail then            ▷ Clear old request
16:            tmp ← ltail
17:            CAS(enqueuers[ltail.enqTid], tmp, null)
18:        end if
19:        for j ← 1 to maxThreads do                         ▷ Find next request
20:            nodeHelp ← enqueuers[(j + ltail.enqTid) mod maxThreads]
21:            if nodeHelp ≠ null then
22:                nullnode ← null
23:                CAS(ltail.next, nullnode, nodeHelp)        ▷ Try append
24:                break
25:            end if
26:        end for
27:        lnext ← ltail.next
28:        if lnext ≠ null then                               ▷ Advance tail if needed
29:            CAS(tail, ltail, lnext)
30:        end if
31:    end for
32:    enqueuers[myIdx] ← null                                ▷ Cleanup if not helped
33:    hp.CLEAR
34: end function
```

is found, the consumer claims it using CAS on the cell's `deq` field in line 23. The slow-path beginning at line 30 publishes dequeue requests that helpers can satisfy by finding unclaimed values or determining the queue is empty. The algorithm maintains linearizability through careful ordering. Enqueues linearize when `T` moves past their cell index, while dequeues linearize when `H` moves past theirs. The helping mechanism ensures that after at most $O(n^2)$ failed attempts, all threads become helpers for a pending operation, guaranteeing completion. [40]

Feldman-Dechev Queue

Uses a sequence number-based mechanism with bitmarking to achieve bounded completion. As shown in algorithms 30 to 32, the ring buffer maintains two atomic counters accessed via `NextTailSeq` in line 7 of algorithm 30 and `NextHeadSeq` in line 7 of algorithm 31. Each position in the buffer array stores either an `ElemNode` containing an element and sequence ID, or an `EmptyNode` containing only a sequence ID. For producers in algorithm 30, threads acquire a sequence ID via FAA on the tail counter in line 7. The position is determined as

Algorithm 24 Turn Queue Dequeue Operation [37]

```
1: function DEQUEUE
2:     myIdx ← GETINDEX
3:     prReq ← deqself[myIdx]                                  ▷ Save previous request
4:     myReq ← deqhelp[myIdx]
5:     deqself[myIdx] ← myReq                                  ▷ Open request
6:     for i ← 0 to maxThreads − 1 do
7:         if deqhelp[myIdx] ≠ myReq then break                ▷ Request satisfied
8:         end if
9:         lhead ← hp.PROTECTPTR(kHpHead, head)
10:        if lhead ≠ head then continue
11:        end if
12:        if lhead = tail then                                ▷ Queue empty
13:            deqself[myIdx] ← prReq                          ▷ Rollback
14:            GIVEUP(myReq, myIdx)
15:            if deqhelp[myIdx] ≠ myReq then                  ▷ Check if helped
16:                deqself[myIdx] ← myReq
17:                break
18:            end if
19:            hp.CLEAR
20:            return null
21:        end if
22:        lnext ← hp.PROTECTPTR(kHpNext, lhead.next)
23:        if lhead ≠ head then continue
24:        end if
25:        if SEARCHNEXT(lhead, lnext) ≠ NOIDX then
26:            CASDEQANDHEAD(lhead, lnext, myIdx)
27:        end if
28:    end for
29:    myNode ← deqhelp[myIdx]
30:    lhead ← hp.PROTECTPTR(kHpHead, head)
31:    if lhead = head and myNode = lhead.next then
32:        CAS(head, lhead, myNode)                            ▷ Help advance head
33:    end if
34:    hp.CLEAR
35:    hp.RETIRE(prReq)                                        ▷ Retire previous node
36:    return myNode.item
37: end function
```

`seqid mod capacity` in line 8. In the common case, threads replace an `EmptyNode` with matching or lower sequence ID with their prepared `ElemNode` via CAS in line 32. The algorithm handles thread delays through backoff and retry mechanisms. If a position contains an `ElemNode` or has a higher sequence ID than assigned, the thread breaks from the inner loop in line 36 to acquire a new sequence ID in the outer loop and attempt insertion at a new position. Bitmarking, setting a flag on a node, is used to indicate positions needing correction by delayed threads in line 20. For consumers in algorithm 31, threads similarly acquire a sequence ID via FAA on the head counter in line 7. They prepare an `EmptyNode` with sequence ID incremented by the buffer capacity in line 9. The dequeue succeeds when replacing an `ElemNode` with matching sequence ID in line 42 via CAS. If encountering an `EmptyNode` or lower sequence ID, threads use backoff and may bitmark `ElemNodes` to maintain FIFO ordering in line 32. The algorithm achieves bounded completion through a progress assurance scheme. After `MAX_FAILS` attempts in line 11, threads post their op-

Algorithm 25 Turn Queue Dequeue Helper Functions [37]

```
1: function SearchNext(lhead, lnext)
2:     turn ← lhead.deqTid
3:     for idx ← turn+1 to turn+maxThreads do
4:         idDeq ← idx mod maxThreads
5:         if deqself[idDeq] ≠ deqhelp[idDeq] then continue
6:         end if
7:         if lnext.deqTid = NOIDX then
8:             CAS(lnext.deqTid, NOIDX, idDeq)                ▷ Assign node
9:         end if
10:        break
11:    end for
12:    return lnext.deqTid
13: end function
14:
15: procedure CasDeqAndHead(lhead, lnext, myIdx)
16:    ldeqTid ← lnext.deqTid
17:    if ldeqTid = myIdx then                                 ▷ My node
18:        deqhelp[ldeqTid] ← lnext                            ▷ Close request
19:    else
20:        ldeqhelp ← hp.ProtectPtr(kHpDeq, deqhelp[ldeqTid])
21:        if ldeqhelp ≠ lnext and lhead = head then
22:            CAS(deqhelp[ldeqTid], ldeqhelp, lnext)          ▷ Help close
23:        end if
24:    end if
25:    CAS(head, lhead, lnext)                                 ▷ Advance head
26: end procedure
27:
28: procedure GiveUp(myReq, myIdx)
29:    lhead ← head
30:    if deqhelp[myIdx] ≠ myReq then return                   ▷ Already helped
31:    end if
32:    if lhead = tail then return                             ▷ Still empty
33:    end if
34:    hp.ProtectPtr(kHpHead, lhead)
35:    if lhead ≠ head then return
36:    end if
37:    lnext ← hp.ProtectPtr(kHpNext, lhead.next)
38:    if lhead ≠ head then return
39:    end if
40:    if SearchNext(lhead, lnext) = NOIDX then
41:        CAS(lnext.deqTid, NOIDX, myIdx)                     ▷ Self-assign
42:    end if
43:    CasDeqAndHead(lhead, lnext, myIdx)
44: end procedure
```

eration to an announcement table and execute a wait-free slow path in lines 12 to 14. The helping mechanism works as follows: `TryHelpAnother` in lines 33 to 45 of algorithm 32 checks one entry in the announcement table per call, cycling through all threads. When an announced operation is found, helpers attempt to complete it using the Associate functions. These functions either claim and complete the operation via CAS or clean up failed attempts by replacing nodes. This ensures every operation completes within $O(N^2_{threads})$ steps, as all threads will eventually help any announced operation. [35]

Algorithm 26 YMC Queue Enqueue Operation [40]

```
1: function ENQUEUE(q, h, v)
2:     for p ← PATIENCE downto 0 do                          ▷ Try fast-path first
3:         if ENQ_FAST(q, h, v, &cell_id) then
4:             return
5:         end if
6:     end for
7:     ENQ_SLOW(q, h, v, cell_id)                            ▷ Fall back to slow-path
8: end function
9:
10: function ENQ_FAST(q, h, v, cid)
11:     i ← FAA(&q → T, 1)                                   ▷ Get unique cell index
12:     c ← FIND_CELL(&h → tail, i)                          ▷ Locate cell, allocate segment if needed
13:     if CAS(c.val, ⊥, v) then                             ▷ Try to deposit value
14:         return true
15:     end if
16:     ∗cid ← i                                             ▷ Return cell index for slow-path
17:     return false
18: end function
19:
20: function ENQ_SLOW(q, h, v, cell_id)
21:     r ← &h → enq.req
22:     r → val ← v                                          ▷ Publish value
23:     r → state ← (1, cell_id)                             ▷ Set pending=1, id=cell_id
24:     tmp_tail ← h → tail
25:     repeat
26:         i ← FAA(&q → T, 1)
27:         c ← FIND_CELL(&tmp_tail, i)
28:         if CAS(c → enq, ⊥e, r) and c.val = ⊥ then        ▷ Reserve cell
29:             TRY_TO_CLAIM_REQ(&r → state, id, i)          ▷ Claim request for this cell
30:             break
31:         end if
32:     until ¬r → state.pending                             ▷ Until helped by dequeuer
33:     id ← r → state.id                                    ▷ Get claimed cell index
34:     c ← FIND_CELL(&h → tail, id)
35:     ENQ_COMMIT(q, c, v, id)                              ▷ Write value to claimed cell
36: end function
```

Verma's Queue

Uses an external helper thread that works on a dedicated core to help other processes to finish their work in a finite number of steps. As shown in algorithm 33, the queue maintains a state array where each worker has a dedicated slot for operation requests. The queue uses a linked list with head and tail pointers managed exclusively by the helper thread. For producers in algorithm 33, threads create a request object containing the operation type and element in line 14, then place it in their designated position in the state array via direct assignment in line 15. They wait until the helper marks the operation as completed in line 16. The algorithm achieves simplicity by delegating all queue modifications to a single helper thread, eliminating the need for complex synchronization. For consumers in algorithm 33, threads similarly create a dequeue request in line 3 and place it in their state array slot in line 4. They wait for completion in line 5, after which the dequeued element is available in the request object in line 9. The helper thread in algorithm 33 continuously traverses the state array in round-robin fashion

Algorithm 27 YMC Queue Enqueue Help Operation [40]

```
1: function HELP_ENQ(q, h, c, i)
2:     if ¬CAS(c → val, ⊥, ⊤) and c → val ≠ ⊤ then                ▷ Value already present
3:         return c → val
4:     end if
5:     if c → enq = ⊥_e then                                      ▷ No request yet, find one to help
6:         repeat
7:             p ← h → enq.peer
8:             r ← &p → enq.req
9:             s ← r → state
10:            if h → enq.id = 0 or h → enq.id = s.id then        ▷ Haven't helped this request
11:                break
12:            end if
13:            h → enq.id ← 0
14:            h → enq.peer ← p → next                            ▷ Move to next peer
15:        until true
16:        if s.pending and s.id ≤ i and ¬CAS(c → enq, ⊥_e, r) then   ▷ Try to reserve cell for peer
17:            h → enq.id ← s.id                                 ▷ Remember we tried to help
18:        else
19:            h → enq.peer ← p → next                           ▷ Peer doesn't need help
20:        end if
21:        if c → enq = ⊥_e then
22:            CAS(c → enq, ⊥_e, ⊤_e)                            ▷ Mark no enqueue will use this cell
23:        end if
24:    end if
25:    if c → enq = ⊤_e then                                      ▷ No enqueue will fill this cell
26:        return (q → T ≤ i ? EMPTY : ⊤)                         ▷ Check if queue was empty
27:    end if
28:    r ← c → enq                                                ▷ Cell has enqueue request
29:    s ← r → state
30:    v ← r → val
31:    if s.id > i then                                           ▷ Request unsuitable for this cell
32:        if c → val = ⊤ and q → T ≤ i then
33:            return EMPTY
34:        end if
35:    else if TRY_TO_CLAIM_REQ(&r → state, s.id, i) or (s = (0, i) and c → val = ⊤) then
36:        ENQ_COMMIT(q, c, v, i)                                 ▷ Help commit the value
37:    end if
38:    return c → val
39: end function
```

in lines 24 to 52. When encountering an enqueue request, it creates a new node in line 29, appends it to the tail in line 30, and updates the tail reference in line 31. For dequeue requests, the helper checks if the queue is empty in line 35 and removes the head element in lines 39 and 40. The helper then updates the request with the dequeued value in line 44. This achieves bounded completion of every process as each operation completes within $O(N)$ steps, where N is the number of workers, since the helper visits each state array position in fixed order. [39]

Wait-Free Circular Queue (wCQ)

The wCQ also uses the fast-path-slow-path by Kogan and Petrank methodology where threads first attempt lock-free operations and fall back to a slow path with helping mechanisms to achieve bounded completion of threads. As shown in algorithms 34 to 36, wCQ extends the

Algorithm 28 YMC Queue Dequeue Operation [40]

```
1: function DEQUEUE(q, h)
2:     for p ← PATIENCE downto 0 do                          ▷ Try fast-path first
3:         v ← DEQ_FAST(q, h, &cell_id)
4:         if v ≠ ⊤ then break
5:         end if
6:     end for
7:     if v = ⊤ then                                         ▷ Fast-path failed
8:         v ← DEQ_SLOW(q, h, cell_id)                       ▷ Use slow-path
9:     end if
10:    if v ≠ EMPTY then                                     ▷ Got value, help peer dequeue
11:        HELP_DEQ(q, h, h → deq.peer)
12:        h → deq.peer ← h → deq.peer → next
13:    end if
14:    return v
15: end function
16:
17: function DEQ_FAST(q, h, id)
18:    i ← FAA(&q → H, 1)                                    ▷ Get unique cell index
19:    c ← FIND_CELL(&h → head, i)
20:    v ← HELP_ENQ(q, h, c, i)                              ▷ Try to get/help produce value
21:    if v = EMPTY then return EMPTY
22:    end if
23:    if v ≠ ⊤ and CAS(c → deq, ⊥_d, ⊤_d) then              ▷ Claim the value
24:        return v
25:    end if
26:    ∗id ← i                                               ▷ Return cell index for slow-path
27:    return ⊤
28: end function
29:
30: function DEQ_SLOW(q, h, cid)
31:    r ← &h → deq.req
32:    r → id ← cid                                          ▷ Set request ID
33:    r → state ← (1, cid)                                  ▷ Publish pending request
34:    HELP_DEQ(q, h, h)                                     ▷ Help complete own request
35:    i ← r → state.idx                                     ▷ Get index where value found
36:    c ← FIND_CELL(&h → head, i)
37:    v ← c → val
38:    ADVANCE_END_FOR_LINEARIZABILITY(&q → H, i + 1)        ▷ Ensure linearizability
39:    return (v = ⊤ ? EMPTY : v)
40: end function
```

lock-free Scalable Circular Queue (sCQ) shown in algorithms 37 and 38 with a variation of Kogan and Petrank's fast-path-slow-path methodology to guarantee wait-freedom. Like sCQ, wCQ uses a ring buffer of size $2n$ while only using n entries at any time, with Head and Tail counters initialized to $2n$ (as in line 32 of algorithm 36). For producers in algorithm 34, threads first help others via `HELP_THREADS` in line 37. The fast path from lines 38 to 44 attempts sCQs `TRY_ENQ` function of algorithm 37 up to `MAX_PATIENCE` times. If unsuccessful, the slow path begins in line 45. The thread records its request in a per-thread descriptor containing the tail value, index, and sequence numbers for integrity checks in lines 46 to 53. It then calls `ENQUEUE_SLOW` in line 54, which repeatedly attempts to insert the element using collaborative synchronization until successful. For consumers in algorithm 34, after checking for empty queue at lines 2 to 4 and helping others at line 5, threads attempt the fast path using `TRY_DEQ`

Algorithm 29 YMC Queue Dequeue Help Operation [40]

```
1: function HELP_DEQ(q, h, helpee)
2:     r ← helpee → deq.req
3:     s ← r → state
4:     id ← r → id
5:     if ¬s.pending or s.idx < id then return                ▷ Request complete or invalid
6:     end if
7:     ha ← helpee → head                                     ▷ Segment pointer for announced cells
8:     s ← r → state                                          ▷ Re-read after getting head
9:     prior ← id; i ← id; cand ← 0
10:    while true do
11:        for hc ← ha; ¬cand and s.idx = prior; do           ▷ Find candidate
12:            c ← FIND_CELL(&hc, ++i)
13:            v ← HELP_ENQ(q, hc, c, i)
14:            if v = EMPTY or (v ≠ ⊤ and c → deq = ⊥_d) then   ▷ Found candidate
15:                cand ← i
16:            else
17:                s ← r → state                              ▷ Check if announced
18:            end if
19:        end for
20:        if cand then
21:            CAS(&r → state, (1, prior), (1, cand))         ▷ Try announce candidate
22:            s ← r → state
23:        end if
24:        if ¬s.pending or r → id ≠ id then return           ▷ Request completed
25:        end if
26:        c ← FIND_CELL(&ha, s.idx)                          ▷ Get announced cell
27:        if c → val = ⊤ or CAS(c → deq, ⊥_d, r) or c → deq = r then
28:            CAS(&r → state, s, (0, s.idx))                 ▷ Complete request
29:            return
30:        end if
31:        prior ← s.idx
32:        if s.idx ≥ i then                                  ▷ Announced cell is ahead
33:            cand ← 0; i ← s.idx                            ▷ Jump forward
34:        end if
35:    end while
36: end function
```

defined in lines 6 to 13 in algorithm 38. On failure, they record their dequeue request in lines 14 to 21 and call `DEQUEUE_SLOW` in line 22, which similarly uses collaborative synchronization to ensure the dequeue completes. Results are gathered from the slow path in lines 25 in 33. The difference to the other queues is the `SLOW_F&A` operation in algorithm 35 at lines 17 to 39, which ensures all cooperative threads (helpee and helpers) increment global counters only once per iteration. The operation works in two phases as seen in line 32 where it atomically updates the global counter with a phase2 pointer using DWCAS (the paper mistakenly calls this DCAS (CAS2)), and line 35 where it clears the `INC` flag. If the system does not support DWCAS, algorithm 39 shows how to substitute it with LL/SC, which can than again be substituted with versioned CAS shown in section 5.2.2. This mechanism allows `ENQUEUE_SLOW` and `DEQUEUE_SLOW` to coordinate multiple threads working on the same operation, ensuring exactly one succeeds while others detect the completion and terminate. Progress is guaranteed through the helping mechanism. `HELP_THREADS` in lines 1 to 15 of algorithm 35 checks one thread per call, cycling through all threads. When finding a pending request, it

Algorithm 30 Feldman-Dechev Queue's Enqueue Operation [35]

```
1: function ENQUEUE(val)
2:     TRYHELPANOTHER
3:     fails ← 0
4:     while true do
5:         if ISFULL then return false
6:         end if
7:         seqid ← NEXTTAILSEQ                                   ▷ FAA on tail
8:         pos ← seqid mod capacity
9:         n_node ← new ElemNode(seqid, val)
10:        while true do
11:            if fails++ = MAX_FAILS then
12:                op ← new EnqueueOp(val)
13:                MAKEANNOUNCEMENT(op)
14:                return op.RESULT
15:            end if
16:            node ← buffer[pos].LOAD
17:            if node.op ≠ null then                            ▷ Operation record
18:                node.op.ASSOCIATE(node, &buffer[pos])
19:                continue
20:            else if ISSKIPPED(node) then                      ▷ Bitmarked
21:                break                                         ▷ Get new seqid
22:            else if node.seqid < seqid then
23:                BACKOFF
24:                if node = buffer[pos].LOAD then
25:                    if ISEMPTYNODE(node) then
26:                        if buffer[pos].CAS(node, n_node) then
27:                            return true
28:                        end if
29:                    end if
30:                end if
31:            else if node.seqid ≤ seqid and ISEMPTYNODE(node) then
32:                if buffer[pos].CAS(node, n_node) then
33:                    return true
34:                end if
35:            else                                              ▷ node.seqid > seqid or ElemNode
36:                break                                         ▷ Get new seqid
37:            end if
38:        end while
39:    end while
40: end function
```

calls `HELP_ENQUEUE` or `HELP_DEQUEUE` in lines 8 and 10. After `MAX_PATIENCE` failed attempts, all threads eventually converge to help stuck threads, ensuring wait-freedom with $O(N^2_{threads})$ complexity. [38]

Excluded but valuable queues

The following two queues were not included because they rely on theoretical hardware primitives that are not available in current hardware. However, they are valuable to know about, since they show how the performance of wait-free queues could be improved with special hardware.

- The queue Khanchandani and Wattenhofer [52] created uses theoretical atomic primitives called half-increment and half-max. Half-increment would be an operation on a theoretical register with two values, first half and second half, that increments the first half if <= second half. Half-max(x) updates the second half to the maximum of its current value and x. The reason this was even introduced was to show that, when combined with CAS, the time complexity of CAS with $O(n)$ would be reduced to $O(\sqrt{n})$, if such hardware would exist.

- Bédin et al. [53] who created a queue using a theoretical atomic primitive called memory-to-memory swap, which changes to memory locations atomically with each other (so a DCAS without the compare part), to show that it would be possible to create wait-free queues as fast as lock-free queues, if the hardware would exist. While thiss is valubale to know how special hardware could improve the performance of wait-free queues, it is not relevant for this thesis, since it is not possible to implement these algorithms on current hardware.

This algorithm was not included in the thesis, because it was running too slow to create valuable benchmark resutls, but still is important to mention, since they show an interesting improvement for time complexity of wait-free queues:

- Naderibeni and Ruppert [54] showed that it is possible to create a wait-free queue using CAS and still achieve a polylogarithmic time complexity by integrating a binary tree structure like Jayanti and Petrovic.

Algorithm 31 Feldman-Dechev Queue's Dequeue Operation [35]

```
1: function DEQUEUE(&result)
2:     TRYHELPANOTHER
3:     fails ← 0
4:     while true do
5:         if ISEMPTY then return false
6:         end if
7:         seqid ← NEXTHEADSEQ                                   ▷ FAA on head
8:         pos ← seqid mod capacity
9:         n_node ← new EmptyNode(seqid + capacity)
10:        while true do
11:            if fails++ = MAX_FAILS then
12:                op ← new DequeueOp
13:                MAKEANNOUNCEMENT(op)
14:                return op.RESULT(result)
15:            end if
16:            node ← buffer[pos].LOAD
17:            if node.op ≠ null then
18:                node.op.ASSOCIATE(node, &buffer[pos])
19:                continue
20:            else if ISSKIPPED(node) and ISEMPTYNODE(node) then
21:                if buffer[pos].CAS(node, n_node) then
22:                    break
23:                end if
24:            else if seqid > node.seqid then                   ▷ Delayed element
25:                BACKOFF
26:                if node = buffer[pos].LOAD then
27:                    if ISEMPTYNODE(node) then
28:                        if buffer[pos].CAS(node, n_node) then
29:                            break
30:                        end if
31:                    else
32:                        SETSKIPPED(&buffer[pos])              ▷ Bitmark
33:                    end if
34:                end if
35:            else if seqid < node.seqid then
36:                break                                         ▷ Get new seqid
37:            else                                              ▷ seqid = node.seqid
38:                if ISELEMNODE(node) then
39:                    if ISSKIPPED(node) then
40:                        n_node ← SETSKIPPED(n_node)
41:                    end if
42:                    if buffer[pos].CAS(node, n_node) then
43:                        result ← node.value
44:                        return true
45:                    end if
46:                else                                          ▷ EmptyNode with matching seqid
47:                    BACKOFF
48:                    if node = buffer[pos].LOAD then
49:                        if buffer[pos].CAS(node, n_node) then
50:                            break
51:                        end if
52:                    end if
53:                end if
54:            end if
55:        end while
56:    end while
57: end function
```

Algorithm 32 Feldman-Dechev Queue's Helper Functions [35]

```
1: function EnqueueOp::Associate(node, address)
2:     success ← helper.CAS(null, node)
3:     if success or helper.Load = node then
4:         node.op.Store(null)                                   ▷ Remove op reference
5:     else
6:         n_node ← new EmptyNode(node.seqid)
7:         if not address.CAS(node, n_node) then
8:             node ← SetSkipped(node)
9:             if address.Load = node then
10:                n_node ← SetSkipped(n_node)
11:                address.CAS(node, n_node)
12:            end if
13:        end if
14:    end if
15: end function
16:
17: function DequeueOp::Associate(node, address)
18:    success ← helper.CAS(null, node)
19:    if success or helper.Load = node then
20:        n_node ← new EmptyNode(node.seqid + capacity)
21:        if not address.CAS(node, n_node) then
22:            node ← SetSkipped(node)
23:            if address.Load = node then
24:                n_node ← SetSkipped(n_node)
25:                address.CAS(node, n_node)
26:            end if
27:        end if
28:    else
29:        node.op.Store(null)
30:    end if
31: end function
32:
33: function TryHelpAnother
34:                                    ▷ Check announcement table and help one operation
35:    helpIdx ← thread-local helping index
36:    op ← announcementTable[helpIdx]
37:    if op ≠ null and op.InProgress then
38:        if op is EnqueueOp then
39:            WaitFreeEnqueue(op)
40:        else
41:            WaitFreeDequeue(op)
42:        end if
43:    end if
44:    helpIdx ← (helpIdx + 1) mod MAX_THREADS
45: end function
```

Algorithm 33 Verma's Queue Operations [39]

```
1: function DEQUEUE
2:     id ← GETTHREADID                                   ▷ Get worker's unique ID
3:     req ← CREATEREQUEST(null, DEQUEUE)                 ▷ Create dequeue request
4:     stateArr[id] ← req                                 ▷ Place request in dedicated slot
5:     while ¬req.isCompleted do                          ▷ Wait for helper to process
6:         wait
7:     end while
8:     stateArr[id] ← null                                ▷ Clear request from state array
9:     return req.e                                       ▷ Return dequeued element
10: end function
11:
12: function ENQUEUE(e)
13:     id ← GETTHREADID                                   ▷ Get worker's unique ID
14:     req ← CREATEREQUEST(e, ENQUEUE)                    ▷ Create enqueue request with element
15:     stateArr[id] ← req                                 ▷ Place request in dedicated slot
16:     while ¬req.isCompleted do                          ▷ Wait for helper to process
17:         wait
18:     end while
19:     stateArr[id] ← null                                ▷ Clear request from state array
20:     return true
21: end function
22:
23: function HELPER
24:     id ← 0                                             ▷ Start at first worker slot
25:     while true do                                      ▷ Continuous helper loop
26:         req ← stateArr[id]                             ▷ Check current slot for request
27:         if req ≠ null ∧ ¬req.isCompleted then          ▷ Found pending request
28:             if req.operation = ENQUEUE then
29:                 n ← new Node(req.e)                    ▷ Create new node
30:                 tail.next ← n                          ▷ Append to tail
31:                 tail ← n                               ▷ Update tail reference
32:                 size ← size + 1
33:                 req.isCompleted ← true                 ▷ Mark request complete
34:             else if req.operation = DEQUEUE then
35:                 if head.next = null then               ▷ Queue is empty
36:                     req.e ← null
37:                     req.isCompleted ← true
38:                 else
39:                     n ← head.next                      ▷ Get first element
40:                     head.next ← n.next                 ▷ Remove from queue
41:                     if n.next = null then              ▷ Queue now empty
42:                         tail ← head                    ▷ Reset tail
43:                     end if
44:                     req.e ← n.e                        ▷ Store dequeued value
45:                     size ← size − 1
46:                     req.isCompleted ← true
47:                 end if
48:             end if
49:         end if
50:         id ← (id + 1) mod workers                      ▷ Round-robin to next slot
51:     end while
52: end function
```

Algorithm 34 wCQ's Operations [38]

```
1: function DEQUEUE_WCQ
2:     if LOAD(&Threshold) < 0 then
3:         return ∅                                                    ▷ Empty
4:     end if
5:     HELP_THREADS
6:                                                                     ▷ Fast path (SCQ)
7:     count ← MAX_PATIENCE
8:     while −−count ≠ 0 do
9:         idx
10:        head ← TRY_DEQ(&idx)
11:        if head = OK then return idx
12:        end if
13:    end while
14:                                                                    ▷ Slow path (wCQ)
15:    r ← &Record[TID]
16:    seq ← r.seq1
17:    r.localHead ← head
18:    r.initHead ← head
19:    r.enqueue ← false
20:    r.seq2 ← seq
21:    r.pending ← true
22:    DEQUEUE_SLOW(head, r)
23:    r.pending ← false
24:    r.seq1 ← seq + 1
25:                                                                    ▷ Get slow-path results
26:    h ← COUNTER(r.localHead)
27:    j ← CACHE_REMAP(h mod 2n)
28:    Ent ← LOAD(&Entry[j].Value)
29:    if Ent.Cycle = CYCLE(h) and Ent.Index ≠ ⊥ then
30:        CONSUME(h, j, Ent)
31:        return Ent.Index
32:    end if
33:    return ∅
34: end function
35:
36: function ENQUEUE_WCQ(index)
37:    HELP_THREADS
38:                                                                    ▷ Fast path (SCQ)
39:    count ← MAX_PATIENCE
40:    while −−count ≠ 0 do
41:        tail ← TRY_ENQ(index)
42:        if tail = OK then return true
43:        end if
44:    end while
45:                                                                    ▷ Slow path (wCQ)
46:    r ← &Record[TID]
47:    seq ← r.seq1
48:    r.localTail ← tail
49:    r.initTail ← tail
50:    r.index ← index
51:    r.enqueue ← true
52:    r.seq2 ← seq
53:    r.pending ← true
54:    ENQUEUE_SLOW(tail, index, r)
55:    r.pending ← false
56:    r.seq1 ← seq + 1
57: end function
```

Algorithm 35 wCQ's Helper Functions [38]

```
1: function HELP_THREADS
2:     r ← &Record[TID]
3:     if −− r.nextCheck ≠ 0 then return
4:     end if
5:     thr ← &Record[r.nextTid]
6:     if thr.pending then
7:         if thr.enqueue then
8:             HELP_ENQUEUE(thr)
9:         else
10:            HELP_DEQUEUE(thr)
11:        end if
12:    end if
13:    r.nextCheck ← HELP_DELAY
14:    r.nextTid ← (r.nextTid + 1) mod NUM_THRDS
15: end function
16:
17: function SLOW_F&A(globalp, local, v, thld)
18:    phase2 ← &Record[TID].phase2
19:    repeat
20:        cnt ← LOAD_GLOBAL_HELP_PHASE2(globalp, local)
21:        if cnt = ∅ or !CAS(local, ∗v, cnt|INC) then
22:            ∗v ← ∗local
23:            if ∗v&FIN then return false
24:            end if
25:            if !(∗v&INC) then return true
26:            end if
27:            cnt ← COUNTER(∗v)
28:        else
29:            ∗v ← cnt|INC
30:        end if
31:        PREPARE_PHASE2(phase2, local, cnt)
32:    until CAS2(globalp, {cnt, null}, {cnt + 1, phase2})
33:    if thld then F&A(thld, −1)
34:    end if
35:    CAS(local, cnt|INC, cnt)
36:    CAS2(globalp, {cnt + 1, phase2}, {cnt + 1, null})
37:    ∗v ← cnt
38:    return true
39: end function
```

Algorithm 36 wCQ's Data Structures [38]

```
1: struct phase2rec_t {
2:     seq1 : int = 1
3:     *local : int
4:     cnt : int
5:     seq2 : int = 0
6: }
7:
8: struct entpair_t {
9:     Note : int = -1
10:    Value : entry_t = { .Cycle=0, .IsSafe=1, .Enq=1, .Index=⊥ }
11: }
12:
13: Entry[2n] : entpair_t
14:
15: struct thrdrec_t {
16:                                                    ▷ === Private Fields ===
17:    nextCheck : int = HELP_DELAY
18:    nextTid : int                                   ▷ Thread ID
19:                                                    ▷ === Shared Fields ===
20:    phase2 : phase2rec_t                            ▷ Phase 2
21:    seq1 : int = 1
22:    enqueue : bool
23:    pending : bool = false
24:    localTail, initTail : int
25:    localHead, initHead : int
26:    index : int
27:    seq2 : int = 0
28: }
29:
30: Record[NUM_THRDS] : thrdrec_t
31: Threshold : int = -1                               ▷ Empty wCQ
32: Tail : int = 2n, Head : int = 2n
```

Algorithm 37 Lock-free Circular Queue (SCQ): Enqueue Operations [38]

```
1:  Threshold : int = -1                                                    ▷ Empty SCQ
2:  Tail : int = 2n, Head : int = 2n
3:                                                  ▷ Init entries: {.Cycle=0, .IsSafe=1, .Index=⊥}
4:  Entry[2n] : entry_t
5:
6:  function ENQUEUE_SCQ(index)
7:      while TRY_ENQ(index) ≠ OK do
8:                                                                          ▷ Try again
9:      end while
10: end function
11:
12: function TRY_ENQ(index)
13:     T ← F&A(&Tail, 1)
14:     j ← CACHE_REMAP(T mod 2n)
15:     E ← LOAD(&Entry[j])
16:     if E.Cycle < CYCLE(T) and (E.IsSafe or LOAD(&Head) ≤ T) and (E.Index = ⊥ or ⊥_c) then
17:         New ← {CYCLE(T), 1, index}
18:         if !CAS(&Entry[j], E, New) then
19:             goto 22
20:         end if
21:         if LOAD(&Threshold) ≠ 3n − 1 then
22:             STORE(&Threshold, 3n − 1)
23:         end if
24:         return OK                                                       ▷ Success
25:     end if
26:     return T                                                            ▷ Try again
27: end function
```

Algorithm 38 Lock-free Circular Queue (SCQ): Dequeue Operations [38]

```
1: function TRY_DEQ(*index)
2:     H ← F&A(&Head, 1)
3:     j ← CACHE_REMAP(H mod 2n)
4:     E ← LOAD(&Entry[j])
5:     if E.Cycle = CYCLE(H) then
6:         CONSUME(H, j, E)
7:         *index ← E.Index
8:         return OK                                          ▷ Success
9:     end if
10:    New ← {E.Cycle, 0, E.Index}
11:    if E.Index = ⊥ or ⊥_c then
12:        New ← {CYCLE(H), E.IsSafe, ⊥}
13:    end if
14:    if E.Cycle < CYCLE(H) then
15:        if !CAS(&Entry[j], E, New) then
16:            goto 33
17:        end if
18:        T ← LOAD(&Tail)                                    ▷ Exit if
19:        if T ≤ H + 1 then                                  ▷ empty
20:            CATCHUP(T, H + 1)
21:        end if
22:        F&A(&Threshold, −1)
23:        *index ← ∅                                         ▷ Empty
24:        return OK                                          ▷ Success
25:    end if
26:    if F&A(&Threshold, −1) ≤ 0 then
27:        *index ← ∅                                         ▷ Empty
28:        return OK                                          ▷ Success
29:    end if
30:    return H                                               ▷ Try again
31: end function
32:
33: function DEQUEUE_SCQ
34:    if LOAD(&Threshold) < 0 then
35:        return ∅                                           ▷ Empty
36:    end if
37:    while TRY_DEQ(&idx) ≠ OK do
38:                                                           ▷ Try again
39:    end while
40:    return idx
41: end function
42:
43: function CONSUME(h, j, e)
44:    OR(&Entry[j], {0, 0, ⊥_c})
45: end function
```

Algorithm 39 CAS2 implementation for wCQ using LL/SC [38]

```
1: function CAS2_VALUE(Var: entpair_t*, Expect: entpair_t, New: entpair_t)
2:     Prev.Value ← LL(&Var → Value)
3:     Prev.Note ← LOAD(&Var → Note)
4:     if Prev ≠ Expect then
5:         return false
6:     end if
7:     return SC(&Var → Value, New.Value)
8: end function
9:
10: function CAS2_NOTE(Var: entpair_t*, Expect: entpair_t, New: entpair_t)
11:     Prev.Note ← LL(&Var → Note)
12:     Prev.Value ← LOAD(&Var → Value)
13:     if Prev ≠ Expect then
14:         return false
15:     end if
16:     return SC(&Var → Note, New.Note)
17: end function
```

6. Implementation

This chapter presents the implementation details of the concurrent queue algorithms in Rust, focusing on the adaptations necessary for inter-process communication (IPC) over shared memory. While the algorithmic logic of each queue has been discussed in the analysis chapter, this implementation required significant engineering to support IPC, ensure cache efficiency, and correctly implement various atomic primitives. The following sections examine how theoretical algorithms were transformed into practical, high-performance implementations suitable for shared memory environments.

6.1. Shared Memory Management for IPC

Inter-process communication through shared memory presents unique challenges compared to traditional heap-based implementations. The primary constraint is that shared memory regions may be mapped at different virtual addresses in different processes, requiring all data structures to be completely position-independent. Additionally, dynamic memory allocation is not possible within shared memory regions, necessitating pre-allocated memory pools for all dynamic data structures.

6.1.1. Memory Layout and Initialization

All queue implementations follow a consistent pattern for shared memory initialization. The most complex example, the WCQueue, demonstrates how multiple components are laid out in shared memory with proper alignment:

```
pub unsafe fn init_in_shared(mem: *mut u8, num_threads: usize) ->
    ↪ &'static mut Self {
    let mut current_offset = 0usize;

    // Align and place queue structure
    current_offset = (current_offset + self_align - 1) & !(self_align -
    ↪ 1);
    let q_ptr = mem.add(current_offset) as *mut Self;
    current_offset += mem::size_of::<Self>();

    // Align and place entry arrays
    current_offset = (current_offset + entry_align - 1) & !(entry_align -
    ↪ 1);
    let aq_entries = mem.add(current_offset) as *mut EntryPair;
    current_offset += capacity * mem::size_of::<EntryPair>();

```

```
    // Initialize structures in-place
    ptr::write(q_ptr, Self {
        aq_entries_offset: current_offset,
        // Store offsets instead of pointers
    });
}
```

Listing 6.1: Memory layout initialization in WCQueue

The alignment calculation `(current_offset + align - 1) & !(align - 1)` ensures that each component starts at a properly aligned address. This is crucial for atomic operations, which often require natural alignment. The queue structure stores offsets relative to the base pointer rather than absolute pointers, ensuring position independence. When accessing these components later, the offset is added to the base pointer:

```
unsafe fn get_entry(&self, entries_offset: usize, idx: usize) ->
    ↪ &EntryPair {
    let entries = self.base_ptr.add(entries_offset) as *const EntryPair;
    &*entries.add(idx)
}
```

Listing 6.2: Position-independent component access

6.1.2. Pre-allocated Memory Pools

Dynamic memory allocation within shared memory is impossible because malloc/free operate on process-private heaps. Therefore, all queues that would normally allocate memory dynamically have been adapted to use pre-allocated pools. The JiffyQueue implementation showcases a sophisticated pool management system with free lists:

```
unsafe fn alloc_node_array_slice(&self) -> *mut Node<T> {
    // Try reuse from free list first
    loop {
        let free_head =
    ↪ self.node_array_slice_free_list_head.load(Ordering::Acquire);
        if free_head.is_null() {
            break;
        }

        let next_free = (*(free_head as *mut AtomicPtr<Node<T>>))
            .load(Ordering::Acquire);

        if self.node_array_slice_free_list_head.compare_exchange(
            free_head,
            next_free,
            Ordering::AcqRel,
            Ordering::Acquire
        ).is_ok() {
            return free_head;
        }
    }
```

```
21
22      // Allocate from pre-allocated pool
23      let nodes_needed = self.buffer_capacity_per_array;
24      let start_idx = self.node_arrays_next_free_node_idx
25          .fetch_add(nodes_needed, Ordering::AcqRel);
26
27      if start_idx + nodes_needed > self.node_arrays_pool_total_nodes {
28          self.node_arrays_next_free_node_idx
29              .fetch_sub(nodes_needed, Ordering::Relaxed);
30          return ptr::null_mut(); // Pool exhausted
31      }
32
33      self.node_arrays_pool_start.add(start_idx)
34  }
```

Listing 6.3: Lock-free memory pool allocation

This implementation maintains a lock-free free list using compare-and-swap operations. When the free list is empty, it falls back to bump allocation from the pre-allocated pool. The `fetch_add` operation atomically increments the allocation index, ensuring thread-safe allocation without locks. If the pool is exhausted, the operation fails gracefully by returning a null pointer.

6.2. Cache Line Optimization

Modern processors transfer data between cores in cache line units (typically 64 bytes). When multiple threads access data in the same cache line, even to different variables, the cache coherence protocol causes the cache line to bounce between cores, significantly degrading performance. This phenomenon, known as false sharing, is particularly problematic in producer-consumer queues.

6.2.1. Explicit Cache Line Padding

The BlqQueue exemplifies how to eliminate false sharing through explicit cache line separation:

```
1  const CACHE_LINE_SIZE: usize = 64;
2
3  #[repr(C)]
4  #[cfg_attr(any(target_arch = "x86_64", target_arch = "aarch64"),
       ↪ repr(align(64)))]
5  pub struct SharedIndices {
6      pub write: AtomicUsize,  // Producer's cache line
7      pub read: AtomicUsize,   // Consumer's cache line
8  }
9
10 #[repr(C, align(64))]
11 struct ProducerPrivate {
```

```
    read_shadow: usize,       // Local copy to avoid false sharing
    write_priv: usize,        // Producer-only write position
}

#[repr(C, align(64))]
struct ConsumerPrivate {
    write_shadow: usize,      // Local copy to avoid false sharing
    read_priv: usize,         // Consumer-only read position
}
```

Listing 6.4: Cache line separation in BlqQueue

The `#[repr(C)]` attribute ensures C-compatible memory layout, while `#[repr(align(64))]` forces the structure to start at a cache line boundary. Although `SharedIndices` contains only two `usize` values (16 bytes total), the alignment ensures they reside in separate cache lines. The producer updates `write` while the consumer updates `read`, preventing false sharing.

The shadow copies (`read_shadow` and `write_shadow`) further reduce cache line transfers. The producer maintains a local copy of the consumer's read position, checking the actual shared variable only when necessary:

```
fn blq_enq_space(&self, needed: usize) -> usize {
    let prod_priv = unsafe { &mut *self.prod_private.get() };

    let mut free_slots = (self.capacity - K_CACHE_LINE_SLOTS)
    ↪ .wrapping_sub(prod_priv.write_priv.wrapping_sub(prod_priv.read_shadow));

    // Lazy load if not enough space
    if free_slots < needed {
        prod_priv.read_shadow =
    ↪ self.shared_indices.read.load(Ordering::Acquire);
        free_slots = (self.capacity - K_CACHE_LINE_SLOTS)
    ↪ .wrapping_sub(prod_priv.write_priv.wrapping_sub(prod_priv.read_shadow));
    }
    free_slots
}
```

Listing 6.5: Shadow variables to reduce cache line transfers

6.2.2. Configurable Padding Strategy

The LamportQueue implements a more flexible approach using Rust's trait system, allowing compile-time selection of padding:

```
pub trait PaddingMode: Send + Sync + 'static {
    type Padding: Send + Sync + Default;
}

```

```
pub struct Unpadded;
pub struct Padded;

pub struct CachePadding {
    _pad: [u8; 120],  // 128 bytes total - 8 for AtomicUsize
}

impl PaddingMode for Padded {
    type Padding = CachePadding;
}

#[repr(C)]
pub struct LamportQueue<T: Send, P: PaddingMode = Unpadded> {
    pub head: AtomicUsize,
    _padding1: P::Padding,  // Configurable cache line separation
    pub tail: AtomicUsize,
    _padding2: P::Padding,
}
```

Listing 6.6: Trait-based configurable padding

This design allows the same queue implementation to be used with or without padding. The benchmarks can test both configurations to measure the impact of false sharing. When instantiated as `LamportQueue<T, Padded>`, the structure expands to place head and tail in separate cache lines. With `LamportQueue<T, Unpadded>`, no padding is inserted, allowing measurement of false sharing effects.

6.2.3. Manual Padding Arrays

For structures where alignment alone is insufficient, manual padding arrays provide precise control:

```
#[repr(C, align(64))]
struct FetchIncrement {
    value: AtomicUsize,
    _padding: [u8; CACHE_LINE_SIZE - std::mem::size_of::<AtomicUsize>()],
}

#[repr(C, align(64))]
struct Node<T> {
    val: Option<T>,
    next: AtomicPtr<Node<T>>,
    _padding: [u8; CACHE_LINE_SIZE - 24], // Fill remaining cache line
}
```

Listing 6.7: Manual padding for exact cache line control

The padding array size is calculated to fill the remainder of the cache line. For `FetchIncrement`, the `AtomicUsize` occupies 8 bytes, so 56 bytes of padding complete the 64-byte cache line. This ensures each `FetchIncrement` instance occupies exactly one cache line, preventing false sharing in arrays of such structures.

6.3. Atomic Primitives Implementation

The correct implementation of atomic primitives is crucial for both correctness and performance. Rust provides a comprehensive set of atomic operations with explicit memory ordering semantics, allowing fine-grained control over synchronization.

6.3.1. Fetch-and-Add (FAA)

Fetch-and-add atomically increments a value and returns the previous value. This operation is fundamental for maintaining counters and indices:

```
// From DQueue - global tail counter with acquire-release ordering
let item_global_location = self.global_tail_location
    .fetch_add(1, Ordering::AcqRel);

// From BQueue - private counter with relaxed ordering
let idx = self.next_item.fetch_add(1, Ordering::Relaxed);

// From WCQueue - with sequential consistency for wait-free algorithm
let seqid = self.tail.fetch_add(1, Ordering::SeqCst);
```

Listing 6.8: Fetch-and-add with different memory orderings

The memory ordering parameter determines synchronization guarantees. `Ordering::Relaxed` provides no synchronization, suitable for private counters. `Ordering::AcqRel` ensures acquire semantics for the read and release semantics for the write, establishing happens-before relationships. `Ordering::SeqCst` provides the strongest guarantees, ensuring a total order across all sequentially consistent operations.

6.3.2. Compare-and-Swap (CAS)

Compare-and-swap atomically updates a value only if it matches an expected value, returning whether the operation succeeded:

```
// Strong CAS with sequential consistency (from WCQueue)
match entry.value.compare_exchange(
    packed,
    new_packed,
    Ordering::SeqCst,    // Success ordering
    Ordering::SeqCst,    // Failure ordering
) {
    Ok(_) => {
        fence(Ordering::SeqCst);  // Additional synchronization
        Ok(())
    }
    Err(current) => {
        // Retry with current value
    }
}

```

```
// Weak CAS for performance (from multiple queues)
entry.value.compare_exchange_weak(
    old_value,
    new_value,
    Ordering::AcqRel,
    Ordering::Acquire,
)
```

Listing 6.9: Compare-and-swap variants and usage patterns

The weak variant (`compare_exchange_weak`) may fail spuriously even when the values match, but can be more efficient on some architectures. The strong variant guarantees success when values match. The two ordering parameters specify synchronization for success and failure cases respectively.

6.3.3. Swap Operations

Swap unconditionally replaces a value and returns the previous value:

```
// From DavidQueue - unconditional slot update
unsafe fn swap(&self, new_val: usize) -> usize {
    self.value.swap(new_val, Ordering::AcqRel)
}

// From TurnQueue - atomic pointer swap
let prev_tail = self.tail.swap(new_node_ptr, Ordering::AcqRel);

// From DrescherQueue - simpler FAS primitive
let prev_tail_ptr = self.tail.swap(new_node_ptr, Ordering::AcqRel);
(*prev_tail_ptr).next.store(new_node_ptr, Ordering::Release);
```

Listing 6.10: Unconditional atomic swap operations

Swap operations are useful when the previous value is needed but the update is unconditional. The DrescherQueue uses swap to implement its simple enqueue operation, atomically updating the tail pointer and then linking the previous tail to the new node.

6.3.4. Load and Store with Memory Ordering

Even simple loads and stores require careful consideration of memory ordering:

```
// Relaxed ordering for private variables
let head = self.head.load(Ordering::Relaxed);
let batch_head = unsafe { *self.batch_head.get() };

// Acquire ordering for reading shared state
let tail = self.tail.load(Ordering::Acquire);
if tail > head {
    // Safe to proceed - acquire ensures we see all writes before tail
    ↪ update
}
```

```

// Release ordering for publishing updates
unsafe { (*slot.data.get()).write(item); }  // Write data first
self.head.store(new_head, Ordering::Release);  // Then publish

// Sequential consistency for strong synchronization
let val = entry.value.load(Ordering::SeqCst);
entry.value.store(new_val, Ordering::SeqCst);
```

Listing 6.11: Memory ordering for loads and stores

The acquire-release pattern is particularly important. A release store synchronizes with an acquire load of the same location, ensuring that all writes before the release store are visible to any thread that sees the acquire load. This pattern is fundamental to the producer-consumer relationship in queues.

6.3.5. Memory Fences

Explicit memory barriers provide synchronization without associated memory operations:

```
// From WCQueue - ensuring visibility across operations
fence(Ordering::SeqCst);
let packed = entry.value.load(Ordering::SeqCst);
let e = EntryPair::unpack_entry(packed);

if condition {
    fence(Ordering::SeqCst);  // Ensure all prior operations complete

    match entry.value.compare_exchange_weak(
        packed,
        new_packed,
        Ordering::SeqCst,
        Ordering::SeqCst,
    ) {
        Ok(_) => {
            fence(Ordering::SeqCst);  // Ensure visibility before
    ↪ proceeding
        }
    }
}

// From DynListQueue - write barrier pattern
fence(Ordering::Release);  // WMB from paper
let current_tail = self.tail.load(Ordering::Acquire);
unsafe {
    (*current_tail).next.store(new_node, Ordering::Release);
}
self.tail.store(new_node, Ordering::Release);
```

Listing 6.12: Explicit memory fence usage

Fences ensure ordering between operations that might not otherwise synchronize. The WCQueue uses sequential consistency fences liberally to ensure correctness in its complex wait-free algorithm. The DynListQueue uses a release fence to implement the write memory barrier specified in the original paper.

6.3.6. Versioned CAS (Simulating LL/SC)

Some algorithms assume Load-Linked/Store-Conditional (LL/SC) primitives, which x86-64 does not provide. The Jayanti-Petrovic queue simulates LL/SC using versioned CAS:

```rust
#[repr(C)]
struct CompactMinInfo {
    version: u32,     // Version counter for ABA prevention
    ts_val: u16,      // Timestamp value (compressed)
    ts_pid: u8,       // Process ID (compressed)
    leaf_idx: u8,     // Leaf index (compressed)
}

impl CompactMinInfo {
    fn to_u64(self) -> u64 {
        ((self.version as u64) << 32)
            | ((self.ts_val as u64) << 16)
            | ((self.ts_pid as u64) << 8)
            | (self.leaf_idx as u64)
    }
}

unsafe fn cas_min_info(&self, old_compact: CompactMinInfo,
                       new_min_info: MinInfo) -> bool {
    // Increment version on every update
    let new_compact = CompactMinInfo::from_min_info(
        new_min_info,
        old_compact.version + 1
    );

    self.compact_min_info.compare_exchange(
        old_compact.to_u64(),
        new_compact.to_u64(),
        Ordering::AcqRel,
        Ordering::Acquire
    ).is_ok()
}
```

Listing 6.13: Versioned CAS for LL/SC simulation

The version counter prevents the ABA problem where a value changes from A to B and back to A between observations. By incrementing the version on every update, even if the logical value returns to a previous state, the version ensures the CAS will fail, simulating LL/SC semantics.

6.3.7. Tagged Pointers

Several queues use pointer tagging to store metadata in unused pointer bits:

```rust
#[repr(transparent)]
struct Slot<T> {
    ptr: AtomicPtr<T>,
}

impl<T> Slot<T> {
    const FLAG_BIT: usize = 0x1;  // Lowest bit as flag

    fn store_data(&self, data_ptr: *mut T) {
        // Set flag bit to indicate "full"
        let tagged_ptr = ((data_ptr as usize) | Self::FLAG_BIT) as *mut T;
        self.ptr.store(tagged_ptr, Ordering::Release);
    }

    fn load_data(&self) -> *mut T {
        let tagged_ptr = self.ptr.load(Ordering::Acquire);
        // Clear flag bit to get real pointer
        ((tagged_ptr as usize) & !Self::FLAG_BIT) as *mut T
    }

    fn is_empty(&self) -> bool {
        let ptr = self.ptr.load(Ordering::Acquire);
        ptr.is_null() || (ptr as usize & Self::FLAG_BIT) == 0
    }
}
```

Listing 6.14: Tagged pointer implementation

On 64-bit systems, pointers typically use only 48 bits, leaving the upper 16 bits available. Additionally, aligned pointers have zeros in their lower bits. The IFFQ queue uses the lowest bit to indicate whether a slot contains data, allowing atomic updates of both pointer and state. The flag bit is set when storing data and cleared when the slot is empty, eliminating the need for a separate state variable.

6.3.8. Atomic Flags for State Management

When pointer tagging is inappropriate, explicit atomic flags provide clear semantics:

```rust
// From FFQ - using AtomicBool for slot state
pub struct Slot<T> {
    pub flag: AtomicBool,  // false = empty, true = full
    _pad: [u8; 8 - std::mem::size_of::<AtomicBool>()],
    data: UnsafeCell<MaybeUninit<T>>,
}

// Producer operation
fn push(&self, item: T) -> Result<(), Self::PushError> {
    let head = self.head.load(Ordering::Relaxed);
```

```
    let slot = self.get_slot(head);

    // Check if slot is available
    if slot.flag.load(Ordering::Acquire) {
        return Err(FfqPushError(item));  // Slot is full
    }

    // Write data
    unsafe { (*slot.data.get()).write(item); }

    // Mark slot as full - linearization point
    slot.flag.store(true, Ordering::Release);

    // Update head
    self.head.store(head.wrapping_add(1), Ordering::Release);
    Ok(())
}
```

Listing 6.15: Atomic boolean flags for state tracking

The FFQ queue demonstrates clear separation between data and metadata. The atomic flag indicates slot state, while the data itself is stored separately. This approach is clearer than pointer tagging and works with any data type, but requires additional memory for the flags.

6.4. Wait-Freedom Adaptations

Wait-free algorithms guarantee that every operation completes within a bounded number of steps, regardless of the behavior of other threads. In shared memory IPC, where process scheduling is unpredictable, wait-freedom is particularly valuable.

6.4.1. Bounded Retry Loops

All potentially unbounded loops must be limited to ensure wait-freedom:

```
unsafe fn help_enqueue(&self, op: *mut EnqueueOp, helper_thread_id:
    ↪ usize) {
    // Bound based on number of threads (paper's requirement)
    let max_help_attempts = MAX_FAILS + (self.num_threads *
    ↪ self.num_threads);

    for attempt in 0..max_help_attempts {
        if (*op).is_complete() {
            return;  // Operation completed
        }

        let node_val = self.get_node(pos).load(Ordering::Acquire);
        let node = Node { value: node_val };

        if node.is_empty() && node.get_seqid() <= seqid {
            // Try to help complete the operation
```

```
            let new_node = Node::new_value(value_ptr, seqid);

            if self.get_node(pos).compare_exchange(
                node_val,
                new_node.value,
                Ordering::AcqRel,
                Ordering::Acquire,
            ).is_ok() {
                (*op).complete_success();
                return;
            }
        }

        // Periodic yielding for fairness
        if attempt % 1000 == 0 {
            std::thread::yield_now();
        } else {
            std::hint::spin_loop();
        }
    }

    // Bounded termination - mark operation as failed
    (*op).complete_failure();
}
```

Listing 6.16: Bounded helping mechanism for wait-freedom

The helping mechanism allows threads to complete operations on behalf of delayed threads. The bound ensures that even if a thread is repeatedly preempted, other threads can only spend a limited time trying to help before giving up. The periodic yielding improves fairness by giving other threads a chance to run.

6.4.2. Adaptive Backoff Strategies

Different contention scenarios require different backoff strategies:

```
// From WCQueue - progressive backoff with sleep
if attempt < 5 {
    std::hint::spin_loop();  // Busy wait for very short delays
} else if attempt < 10 {
    fence(Ordering::SeqCst);
    std::thread::yield_now();  // Yield CPU to other threads
} else {
    fence(Ordering::SeqCst);
    std::thread::sleep(Duration::from_micros(1));  // Sleep for longer
    ↪ delays
}

// From BQueue - exponential backoff
let mut spins = 1;
for i in 0..10 {
```

```
    for _ in 0..spins.min(1024) {
        std::hint::spin_loop();
    }
    spins *= 2;  // Double spin count each iteration

    let current = self.get_node(pos).load(Ordering::Acquire);
    if current != expected {
        return true;  // Value changed
    }
}
```

Listing 6.17: Adaptive backoff strategies

The backoff strategy significantly impacts performance. Short spin loops (`spin_loop`) keep the CPU busy but allow immediate response to changes. Yielding (`yield_now`) allows other threads to run but incurs context switch overhead. Sleeping reduces CPU usage but increases latency. The progressive strategy starts with spinning and escalates to sleeping only under high contention.

6.5. Memory Safety Considerations

Implementing lock-free data structures in Rust requires extensive use of unsafe code, as the borrow checker cannot verify the complex ownership patterns inherent in concurrent algorithms.

6.5.1. Shared Memory Size Calculation

Each queue provides a method to calculate the exact shared memory size required:

```
// From BQueue - simple calculation
pub const fn shared_size(capacity: usize) -> usize {
    mem::size_of::<Self>()                                  // Queue structure
        + capacity * mem::size_of::<MaybeUninit<T>>()       // Data storage
        + capacity * mem::size_of::<AtomicBool>()           // Validity flags
}

// From WCQueue - complex layout calculation
fn layout(num_threads: usize, num_indices: usize) -> (Layout, [usize; 4])
    ↪ {
    let ring_size = num_indices.next_power_of_two();
    let capacity = ring_size * 2;

    let root = Layout::new::<Self>();
    let (l_aq_entries, o_aq_entries) = root
        .extend(Layout::array::<EntryPair>(capacity).unwrap())
        .unwrap();
    let (l_fq_entries, o_fq_entries) = l_aq_entries
        .extend(Layout::array::<EntryPair>(capacity).unwrap())
        .unwrap();
```

```
    let (l_records, o_records) = l_fq_entries
        .extend(Layout::array::<ThreadRecord>(num_threads).unwrap())
        .unwrap();
    let (l_final, o_data) = l_records
        .extend(Layout::array::<DataEntry<T>>(num_indices).unwrap())
        .unwrap();

    (l_final.pad_to_align(), [o_aq_entries, o_fq_entries, o_records,
    ↪ o_data])
}
```

Listing 6.18: Shared memory size calculation methods

The size calculation must account for all components including alignment padding. The `Layout` API ensures proper alignment and calculates offsets automatically. For simple queues, manual calculation suffices. Complex queues with multiple components benefit from the `Layout` API's automatic offset calculation.

6.5.2. Unsafe Code Patterns

The implementation follows strict patterns for unsafe code:

```
// In-place initialization pattern
ptr::write(queue_ptr, Self {
    head: AtomicUsize::new(0),
    tail: AtomicUsize::new(0),
    /* other fields */
});

// Careful pointer arithmetic with bounds checking
let slot_idx = current_write & self.mask;  // Ensure within bounds
let slot_ptr = self.buffer.add(slot_idx);  // Safe offset

// Manual memory management with explicit initialization
let item = ptr::read(&(*node_ptr).data).assume_init();  // Read
    ↪ initialized data
ptr::write(node_ptr, MaybeUninit::new(item));  // Write through
    ↪ MaybeUninit

// Atomic pointer dereferencing pattern
let node = self.head.load(Ordering::Acquire);
if !node.is_null() {
    let next = unsafe { (*node).next.load(Ordering::Acquire) };
    // Use next...
}
```

Listing 6.19: Common unsafe code patterns

These patterns ensure memory safety despite the unsafe blocks. Pointer arithmetic uses masking to prevent out-of-bounds access. The `MaybeUninit` type explicitly tracks initialization state. Null checks prevent dereferencing invalid pointers. While the compiler cannot verify these patterns, consistent application reduces the likelihood of memory safety violations.

7. Benchmarking and Results

8. Conclusion and Future Work

Bibliography

[1] M. Herlihy, "Wait-free synchronization", *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991. DOI: 10.1145/114005.102808. [Online]. Available: https://doi.org/10.1145/114005.102808.

[2] B. B. Brandenburg, *Multiprocessor real-time locking protocols: A systematic review*, 2019. arXiv: 1909.09600 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1909.09600.

[3] O. Kode and T. Oyemade, *Analysis of synchronization mechanisms in operating systems*, 2024. arXiv: 2409.11271 [cs.OS]. [Online]. Available: https://arxiv.org/abs/2409.11271.

[4] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures", *SIGPLAN Not.*, vol. 47, no. 8, pp. 141–150, Feb. 2012. DOI: 10.1145/2370036.2145835. [Online]. Available: https://doi.org/10.1145/2370036.2145835.

[5] S. Timnat and E. Petrank, "A practical wait-free simulation for lock-free data structures", *SIGPLAN Not.*, vol. 49, no. 8, pp. 357–368, Feb. 2014. DOI: 10.1145/2692916.2555261. [Online]. Available: https://doi.org/10.1145/2692916.2555261.

[6] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms", in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275. DOI: 10.1145/248052.248106. [Online]. Available: https://doi.org/10.1145/248052.248106.

[7] H. Huang, P. Pillai, and K. G. Shin, "Improving Wait-Free algorithms for interprocess communication in embedded Real-Time systems", in *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA: USENIX Association, Jun. 2002. [Online]. Available: https://www.usenix.org/conference/2002-usenix-annual-technical-conference/improving-wait-free-algorithms-interprocess.

[8] A. Pellegrini and F. Quaglia, *On the relevance of wait-free coordination algorithms in shared-memory hpc:the global virtual time case*, 2020. arXiv: 2004.10033 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2004.10033.

[9] B. Xu, B. Chu, H. Fan, and Y. Feng, "An analysis of the rust programming practice for memory safety assurance", in *Web Information Systems and Applications: 20th International Conference, WISA 2023, Chengdu, China, September 15–17, 2023, Proceedings*, Chengdu, China: Springer-Verlag, 2023, pp. 440–451. DOI: 10.1007/978-981-99-6222-8_37. [Online]. Available: https://doi.org/10.1007/978-981-99-6222-8_37.

[10] A. Sharma, S. Sharma, S. Torres-Arias, and A. Machiry, *Rust for embedded systems: Current state, challenges and open problems (extended report)*, 2024. arXiv: 2311.05063 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2311.05063.

[11] K. Kavi, R. Akl, and A. Hurson, "Real-time systems: An introduction and the state-of-the-art", in Mar. 2009. DOI: 10.1002/9780470050118.ecse344.

[12] X. Hua, J. Zeng, H. Li, J. Huang, M. Luo, X. Feng, H. Xiong, and W. Wu, "A review of automobile brake-by-wire control technology", *Processes*, vol. 11, p. 994, Mar. 2023. DOI: 10.3390/pr11040994.

[13] *Inter process communication (ipc)*, https://www.geeksforgeeks.org/inter-process-communication-ipc/, 2025.

[14] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms", 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:6899525.

[15] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors", *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991. DOI: 10.1145/103727.103729. [Online]. Available: https://doi.org/10.1145/103727.103729.

[16] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism", *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 95–109, Sep. 1991. DOI: 10.1145/121133.121151. [Online]. Available: https://doi.org/10.1145/121133.121151.

[17] A. Thakur, *Race condition, synchronization, atomic operations and volatile keyword.* https://opensourceforgeeks.blogspot.com/2014/01/race-condition-synchronization-atomic.html, 2014.

[18] *Managing mutual exclusion mechanism for real-time applications*, https://realtimepartner.com/articles/mutual-exclusion.html, 2016.

[19] S. Raghunathan, "Extending inter-process synchronization with robust mutex and variants in condition wait", in *2008 14th IEEE International Conference on Parallel and Distributed Systems*, 2008, pp. 121–128. DOI: 10.1109/ICPADS.2008.98.

[20] *Introduction of deadlock in operating system*, https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/, 2025.

[21] P. A. Buhr, "Concurrency errors", in *Understanding Control Flow: Concurrent Programming Using μC++*. Cham: Springer International Publishing, 2016, pp. 395–423. DOI: 10.1007/978-3-319-25703-7_8. [Online]. Available: https://doi.org/10.1007/978-3-319-25703-7_8.

[22] P. Chahar and S. Dalal, "Deadlock resolution techniques: An overview", *International Journal of Scientific and Research Publications*, vol. 3, no. 7, pp. 1–5, 2013.

[23] Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin, "Solving the group priority inversion problem in a timed asynchronous system", *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 900–915, 2002. DOI: 10.1109/TC.2002.1024738.

[24] M. Michael and M. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms", *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, Mar. 1996. DOI: 10.1145/248052.248106.

[25] F. Hoseini, A. Atalar, and P. Tsigas, "Modeling the performance of atomic primitives on modern architectures", in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19, Kyoto, Japan: Association for Computing Machinery, 2019. DOI: 10.1145/3337821.3337901. [Online]. Available: https://doi.org/10.1145/3337821.3337901.

[26] T. Fuchs and H. Murakami, "Evaluation of task scheduling algorithms and wait-free data structures for embedded multi-core systems", 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:218073330.

[27] M. David, "A single-enqueuer wait-free queue implementation", in *Distributed Computing*, R. Guerraoui, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 132–143.

[28] G. Drescher and W. Schröder-Preikschat, "An experiment in wait-free synchronisation of priority-controlled simultaneous processes: Guarded sections", Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Tech. Rep. CS-2015-01, Jan. 2015. [Online]. Available: https://www4.cs.fau.de/Publications/2015/drescher_15_cstr.pdf.

[29] I. Culic, A. Vochescu, and A. Radovici, "A low-latency optimization of a rust-based secure operating system for embedded devices", *Sensors*, vol. 22, no. 22, p. 8700, 2022.

[30] L. Lamport, "Concurrent reading and writing", *Commun. ACM*, vol. 20, no. 11, pp. 806–811, Nov. 1977. DOI: 10.1145/359863.359878. [Online]. Available: https://doi.org/10.1145/359863.359878.

[31] L. Lamport, "Specifying concurrent program modules", *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 190–222, Apr. 1983. DOI: 10.1145/69624.357207. [Online]. Available: https://doi.org/10.1145/69624.357207.

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

[33] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers", *SIGPLAN Not.*, vol. 46, no. 8, pp. 223–234, Feb. 2011. DOI: 10.1145/2038037.1941585. [Online]. Available: https://doi.org/10.1145/2038037.1941585.

[34] S. Feldman and D. Dechev, "A wait-free multi-producer multi-consumer ring buffer", *SIGAPP Appl. Comput. Rev.*, vol. 15, no. 3, pp. 59–71, Oct. 2015. DOI: 10.1145/2835260.2835264. [Online]. Available: https://doi.org/10.1145/2835260.2835264.

[35] D. Dechev, S. Feldman, and A. Barrington, "A scalable multi-producer multi-consumer wait-free ring buffer.", Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Apr. 2015. [Online]. Available: https://www.osti.gov/biblio/1531271.

[36] A. Barrington, S. Feldman, and D. Dechev, "A scalable multi-producer multi-consumer wait-free ring buffer", in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1321–1328. DOI: 10.1145/2695664.2695924. [Online]. Available: https://doi.org/10.1145/2695664.2695924.

[37] P. Ramalhete and A. Correia, "Poster: A wait-free queue with wait-free memory reclamation", *SIGPLAN Not.*, vol. 52, no. 8, pp. 453–454, Jan. 2017, Full version available at https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturnqueue-2016.pdf. DOI: 10.1145/3155284.3019022. [Online]. Available: https://doi.org/10.1145/3155284.3019022.

[38] R. Nikolaev and B. Ravindran, "Wcq: A fast wait-free queue with bounded memory usage", in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 461–462. DOI: 10.1145/3503221.3508440. [Online]. Available: https://doi.org/10.1145/3503221.3508440.

[39] M. Verma, "Scalable and performance-critical data structures for multicores", Jun. 2013, Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering. [Online]. Available: https://web.tecnico.ulisboa.pt/~ist14191/repository/Thesis_Mudit_Verma.pdf.

[40] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add", in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, Spain: Association for Computing Machinery, 2016. DOI: 10.1145/2851141.2851168. [Online]. Available: https://doi.org/10.1145/2851141.2851168.

[41] J. Wang, Q. Jin, X. Fu, Y. Li, and P. Shi, "Accelerating wait-free algorithms: Pragmatic solutions on cache-coherent multicore architectures", *IEEE Access*, vol. 7, pp. 74 653–74 669, 2019. DOI: 10.1109/ACCESS.2019.2920781.

[42] D. Adas and R. Friedman, "A fast wait-free multi-producers single-consumer queue", in *Proceedings of the 23rd International Conference on Distributed Computing and Networking*, ser. ICDCN '22, Delhi, AA, India: Association for Computing Machinery, 2022, pp. 77–86. DOI: 10.1145/3491003.3491004. [Online]. Available: https://doi.org/10.1145/3491003.3491004.

[43] D. Adas and R. Friedman, "Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue", *CoRR*, vol. abs/2010.14189, 2020. arXiv: 2010.14189. [Online]. Available: https://arxiv.org/abs/2010.14189.

[44] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks", in *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, S. Sarukkai and S. Sen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 408–419.

[45] M. Torquati, *Single-producer/single-consumer queues on shared cache multi-core systems*, 2010. arXiv: 1012.1824 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1012.1824.

[46] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient synchronisation mechanism for multi-core systems", in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., ser. Lecture Notes in Computer Science, The provided PDF indicates this work is related to Euro-Par 2012. Exact page numbers for this specific title within the proceedings were not found in the immediate search; the closely related paper "An Efficient Unbounded Lock-Free Queue for Multi-core Systems" by the same authors appears on pages 662-673 of the same volume., vol. 7484, Berlin, Heidelberg: Springer, 2012.

[47] J. Wang, K. Zhang, X. Tang, and B. Hua, "B-queue: Efficient and practical queuing for fast core-to-core communication", *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 137–159, Feb. 2013. DOI: 10.1007/s10766-012-0213-x. [Online]. Available: https://doi.org/10.1007/s10766-012-0213-x.

[48] V. Maffione, G. Lettieri, and L. Rizzo, "Cache-aware design of general-purpose single-producer–single-consumer queues", *Software: Practice and Experience*, vol. 49, no. 5, pp. 748–779, 2019. DOI: https://doi.org/10.1002/spe.2675. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2675. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2675.

[49] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue", Jan. 2008. DOI: 10.1145/1345206.1345215.

[50] D. B. Demir, `https://github.com/DevrimBaran/MA.git`.

[51] U. Drepper, "What every programmer should know about memory", *Red Hat, Inc*, vol. 11, no. 2007, p. 2007, 2007.

[52] P. Khanchandani and R. Wattenhofer, "On the importance of synchronization primitives with low consensus numbers", in *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ser. ICDCN '18, Varanasi, India: Association for Computing Machinery, 2018. DOI: `10.1145/3154273.3154306`. [Online]. Available: `https://doi.org/10.1145/3154273.3154306`.

[53] D. Bédin, F. Lépine, A. Mostéfaoui, D. Perez, and M. Perrin, "Wait-free algorithms: The burden of the past", Mar. 2024. DOI: `10.21203/rs.3.rs-4125819/v1`.

[54] H. Naderibeni and E. Ruppert, "A wait-free queue with polylogarithmic step complexity", in *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, ser. PODC '23, Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 124–134. DOI: `10.1145/3583668.3594565`. [Online]. Available: `https://doi.org/10.1145/3583668.3594565`.

List of Acronyms

IPC Inter-Process Communication

HRTS Hard Real-Time System

SRTS Soft Real-Time System

RTOS Real-Time Operating System

RTS Real-Time System

CAS Compare and Swap

DWCAS Double-With Compare and Swap

DCAS Double Compare and Swap

FIFO First In First Out

LIFO Last In First Out

MPMC Multi Producer Multi Consumer

MPSC Multi Producer Single Consumer

SPMC Single Producer Multi Consumer

SPSC Single Producer Single Consumer

FAA Fetch and Add

FAS Fetch and Store

LL/SC Load-Linked and Store-Conditional

LL Load-Linked

SC Store-Conditional

VL Validate-Link

FFQ FastForward Queue

IFFQ Improved FastForward Queue

BIFFQ Batched Improved FastForward Queue

uSPSC Unbounded Single Producer Single Consumer

dSPSC Dyncamic Single Producer Single Consumer

mSPSC MultiPush Single Producer Single Consumer

BLQ Batched Lamport Queue

LLQ Lazy Lamport Queue

BLQ Batched Lamport Queue

wCQ Wait-Free Circular Queue

YMC Yang Mellor-Crummey

sCQ Scalable Circular Queue

List of Figures

List of Tables

List of Algorithms

Listings

A. Appendix