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 21, 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 21, 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 Deadlocks, aushungernden Prozessen oder Prioritätsinversionen führen können, welche zu unvorhersehbaren Antwortzeiten führen. Um diese Herausforderungen zu bewältigen, bietet die wait-free Synchronisation eine Alternative, die den Abschluss von Operationen, wie dem Austausch von Daten zwischen mehreren Prozessen, in einer begrenzten Anzahl von Schritten garantiert und so die Systemreaktionsfähigkeit und -vorhersehbarkeit 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 von Synchronisationsmechanismen. Diese Arbeit analysiert, implementert und validiert bestehende wait-free Methoden für IPC über geteilte Speicherregionen in Echtzeitsystemen und vergleicht die Leistung aller gefundenen Methoden in einem IPC über geteilte Speicherregionen Setting miteinander anhand Ihrer Geschwindigkeit.

Stichwörter: Echtzeitsysteme, wait-free Synchronisation, 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 synchronisation mechanisms introduce blocking, which can result in deadlocks, process starvation, or priority inversion, leading to unpredictable response times. To overcome these challenges, wait-free synchronisation provides an alternative that guarantees operation completion, such as the completion of data exchange between multiple processes, within a bounded number of steps, thereby ensuring system responsiveness and predictability.

This thesis examines the application of wait-free data structures for IPC in RTS, with a focus on their implementation in the Rust programming language. Rust's ownership model and strict concurrency guarantees make it well-suited for developing synchronisation mechanisms. This work analyses, implements and validates existing wait-free techniques for real-time IPC, and compares their performance against each other in an IPC over shared memory setting based on their execution times to choose the most optimal algorithms suitable for IPC over shared memory in real-time systems.

Keywords: real-time systems, wait-free synchronisation, lock-free synchronisation, 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, such as 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 synchronisation, problems such as data corruption or race conditions can occur, leading to unpredictable behaviour. Traditional synchronisation methods with locks are commonly used to manage access to shared resources by blocking processes, allowing only one process at a time to access the shared resource and exchange data properly. However, these blocking mechanisms introduce difficulties in real-time settings. Since traditional synchronisation methods require processes to wait for resource availability, they can lead to unpredictable response times through potential deadlocks, process starvation, or priority inversion. Such unpredictability is unacceptable in such systems that require strict timing guarantees. [1]–[3]

To overcome these limitations, synchronisation techniques without blocking mechanisms are required. A lock-free algorithm, for instance, functions without any locking mechanism, thus avoiding blocking. This guarantees that at least one process completes in a finite number of steps, regardless of contention (when multiple processes attempt to access the same shared resource). This property ensures that the system will still function even if one process is lagging. The only problem is that this does not prevent starvation, since there is no guarantee that every process will finish its task. [4]

While lock-free algorithms represent an improvement, wait-free algorithms guarantee that every operation completes in a finite number of steps, regardless of contention. This property ensures system responsiveness and predictability, which are essential for defining timing constraints in real-time applications. [1], [2], [4]

These synchronisation mechanisms are particularly important in the context of IPC, which is needed in RTS. IPC allows processes to exchange data efficiently, but its performance is heavily influenced by the synchronisation techniques used. Traditional IPC mechanisms, which often rely on blocking some processes. Wait-free data structures offer a promising alternative by ensuring that communication operations complete within predictable time bounds. [5]–[8]

To implement the earlier addressed synchronisation techniques properly, the choice of programming language is essential. The Rust programming language provides helpful features for implementing real-time synchronisation mechanisms. Its ownership model and strict type system prevent data races and enforce safe concurrency. Additionally, Rust offers precise control over system resources, making it a suitable choice for real-time applications that require both low latency and high reliability. [9], [10]

The concepts and methods introduced here, including RTS, IPC, synchronisation techniques and their difficulties, wait-free synchronisation, and the Rust programming language, are explored in greater depth in chapter 2.

1.2. Objective

The primary goal of this research is to identify the most effective wait-free data structures for implementing wait-free synchronisation in IPC through shared memory in RTS using Rust. To do so, this study aims to:

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

1.3. Structure of the Thesis

To describe how to achieve this goal, a deeper knowledge base will be provided in chapter 2 to facilitate understanding of the concepts used in this work. Then, in chapter 3, related work leading to concepts needed for this work will be presented. After that, in chapter 4, the methodology for finding the papers, including the wait-free queues, will be explained. Next in chapter 5, it will be explained which data structure was chosen, and the wait-free queues found will be explained. Afterwards, in chapter 6, it will be explained how to implement the essential details of the queues without explaining their logic again and following that, in chapter 7 the results of the benchmarks will be presented and analysed, and finally, in chapter 8 a conclusion will be drawn and future work will be discussed.

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 synchronisation techniques, with a particular focus on wait-free synchronisation. Additionally, the Rust programming language will be examined, as it serves as the primary development environment for this study. Furthermore, existing synchronisation methods in RTS will be explored to contextualise the motivation and contributions of this work.

2.1. Real-Time Systems

In RTS, the correctness of the system depends not only 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. The system must guarantee that every timing constraint is met. A use case in industrial automation is where all machines and robotic modules must communicate with each other as quickly as possible to ensure the manufacturing line is not blocked. [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. In terms of infrastructure, 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 is considered acceptable to drop frames occasionally to ensure the video stream is maintained. [11]

Sometimes these two systems appear in combination, where some functions have hard real-time constraints and some have soft real-time constraints. Krishna et al. provide a good example in their paper where they describe that for the Apollo 11 mission, some components for the landing processes had soft real-time behaviour, and the rest still functioned with hard real-time constraints. [11]

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

2.2. Inter-Process Communication (IPC)

Processes used in an RTS also need to share information with each other so that the system can function. So some kind of IPC is required. IPC enables processes to exchange information with each other using various methods, such as shared memory regions, which are discussed in more detail later in this thesis [12]. In general, IPC is necessary in all computing systems, as processes often need to work together (e.g., a producer process passes data to a consumer process) [12]. Let's take the brake-by-wire technology as an example. Brake-by-wire is a technology for driverless cars, where some mechanical and hydraulic components from the braking system are replaced by wires to transmit braking signals, as there is no longer a driver to press the brake pedal [13]. This, of course, requires different processes to share information. In the context of this thesis, this kind of communication requires strict timing constraints as stated before, since any kind of blockage in a brake-by-wire system could lead to a catastrophe.

2.2.1. Shared Memory

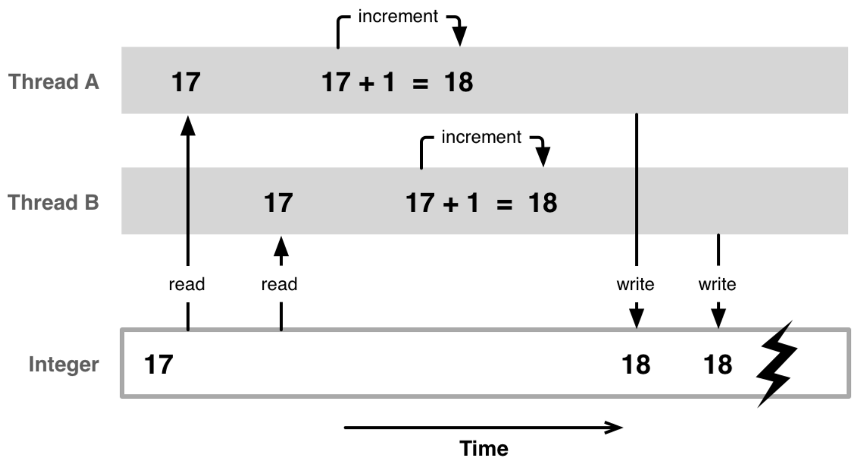
To facilitate information sharing between processes, these processes must have regular access to the same data. With a shared memory segment, multiple processes can have access to the same memory location. Therefore, all processes that are part of the IPC can read and write to this common memory space, thereby avoiding unnecessary data copies. With that, processes exchange information by directly manipulating memory. This kind of IPC is particularly useful for real-time applications, which handle large volumes of data or are required to quickly transfer data between sensors and control tasks. It is also important to note that the section of code that manages these data accesses by different processes is called the critical section. The problem with this is that the system must somehow manage how the processes access the shared memory segment. This is mostly achieved by using various synchronisation techniques. Without any synchronisation mechanism, race conditions or inconsistent data can occur. [14]

2.3. Synchronisation

Synchronisation is essential for 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 correctly synchronised. Traditional synchronisation techniques ensure mutual exclusion (only one process at a time uses the shared resource), thus avoiding race conditions and ensuring data consistency. Race conditions occur when, for example, two processes attempt to write to the same resource. Let's consider a single counter instance with a value of 17 as a shared resource in a shared memory region. If one process, p1, and one process, p2, increment 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 in more detail, fig. 2.1 visualises a race condition with threads.

The primary difference between processes and threads is that threads are part of a process, which can perform multiple tasks simultaneously via threads 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. Thus, a process cannot naturally access the memory of another process. The concept illustrated in fig. 2.1 can still be applied to processes. [15]

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



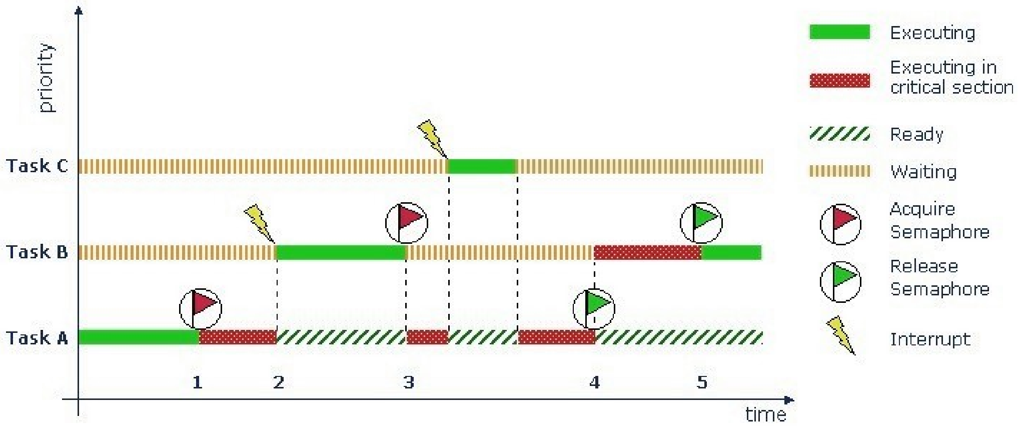
[16]

2.3.1. Mutual Exclusion

As discussed, mutual exclusion only allows 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 tries to access that shared resource x has to wait until the process p1 finishes its task, where it needs that shared resource x. To achieve this, synchronisation techniques based on locks or semaphores are typically used to block the entry of a process into a shared resource that is already accessed and in use by another process. See fig. 2.2 to gain a deeper understanding of how this works. This paper will not go into the details of how traditional synchronisation techniques, such as locks or semaphores, work, as it is only important for this work that these methods manage process access to shared resources in shared memory via some form of locks. A process acquires a lock to access a shared resource and releases it when its task is complete. Another process attempting to access the same resource while it's in use must wait until the lock is released for that resource.

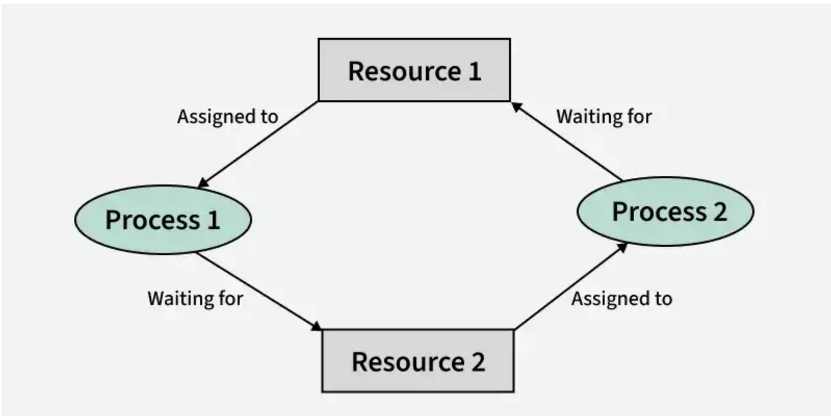
This approach inherently relies on blocking processes. This may lead to several issues, including deadlocks, process starvation, or priority inversion, resulting in unpredictable response times and inability to define timing constraints for RTS [2]. The sequence in which processes might acquire the lock first to enter the critical section, when multiple processes wait for access, is mainly set by a scheduler [2]. Since wait-free methods, as explained later in section 2.5, are lock-free, a scheduler is not required, and therefore, scheduling will not be described in more detail in this work. The problems mentioned above will be discussed in the following sub-subsections:

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



[17]

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



[18]

Process Starvation

What happens when multiple processes attempt to access a shared resource one after another, and one process repeatedly fails to acquire a lock to enter the critical section? This process would wait for an indefinite time and would never enter the shared resource, a condition known as process starvation. This usually happens when a synchronisation method allows one or more processes to make progress while ignoring a particular process or processes. This mostly occurs in environments where some form of process prioritisation exists and processes are classified into low- and high-priority processes. When there are always a lot of high-priority and some low-priority processes available, it might happen that these low-priority processes will never be able to enter the critical section. This is a problem, as these low-priority processes may also be important for the system. [19]

Deadlock

Even worse, what if two or more processes have already accessed a resource and now each waits for the other to release the lock for the resource they acquired so that they can further work? This results in a situation seen in fig. 2.3 where these processes now indefinitely wait for each other and never terminate. Thus, the resources held by these waiting processes are never released and are therefore never available to the other process. As one can see, this brings the system into a state that would prevent any further progress and would no longer respond to any command. [20]

For instance, if a driverless car with a brake-by-wire system is in a deadlock, the vehicle may eventually be unable to brake when needed, resulting in a fatal collision.

Priority Inversion

Now, let's say no process starvations or deadlocks occur. What could also happen is that a lower-priority process already accessed a shared resource, and after that, a higher-priority process needs to access that specific resource too. If the lower-priority process is delayed, the higher-priority process is also delayed. This is known as priority inversion, where a low-priority process delays a high-priority process. cite priorityInversion [21]

2.4. Lock-Free Synchronisation

Due to the problems that traditional synchronisation techniques introduce, synchronisation techniques are required that do not block processes with any kind of locking mechanism. One way could be the implementation of lock-free synchronisation techniques. This would allow multiple processes to access the shared resource concurrently. Lock-free synchronisation ensures that at least one process will complete its task in a finite number of steps. However, some processes may be unable to proceed because lock-free synchronisation does not guarantee that all processes will complete their operations in a finite number of steps. This means that starvation or even priority inversion is still possible, as some processes, even high-priority processes, may be indefinitely delayed. There are various mechanisms to achieve this. One way to achieve lock-freedom, for example, is the lock-free technique introduced by Michael and Scott, which also forms the basis for some other wait-free algorithms.

2.4.1. Michael and Scott's Lock-Free Queue

Michael and Scott developed an algorithm, as shown in algorithm 1, using a linked list as a shared data structure with enqueue and dequeue functions to introduce lock-freedom. A linked list is a list containing nodes that contain data and a pointer called 'next', which references the next node in the list. This list 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 called 'tail', which references the end node of the list. The core concept of the algorithm is the enqueue and dequeue functions, which begin at lines 7 and 26 in algorithm 1, used to add and remove nodes from the shared data structure. When a process tries to add a node to the list, it first creates a new node and sets its next pointer to `NULL`, as seen on lines 8 to 10 of the enqueue function. Beginning from line 11 to line 23, the following occurs: The process first checks if the pointer referencing the next node after the tail node is `NULL`, as seen on line 15. If it is `NULL`, it tries to link the new node to the end of the list by using a 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. [22]

So let's say 2 processes, p1 and p2, execute up to line 16 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 does not execute further, thus not finalising the enqueue with line 24 and p2 retries the loop until line 15, the condition in line 15 would not be `TRUE` anymore for p1 and p1 would execute lines 19 and 20 to help p2 to finalise its enqueue so other processes can work further with this algorithm. [22]

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 that could not finish its enqueue would cause confusion for other processes in the dequeue function, the process that could not finish its enqueue will also be helped in the dequeue function. [22]

Initialisation starts at line 1 in algorithm 1, which is used solely to create dummy nodes when there are no nodes 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. However, this approach has one major problem. If, for instance, process p1 is trying to enqueue, it can happen that the CAS loop might fail indefinitely if, for an indefinite time, other processes are always executing line 16 immediately before p1 could execute line 16. This means that in very high contention scenarios, a process may be delayed indefinitely and starve out. In an HRTS, this could lead to a violation of timing constraints, as the process would not complete its task within the defined timing window, which is unacceptable. This is why a slightly different approach is needed, one that guarantees every process will complete its operation in a finite number of steps. [22]

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. An overview of the atomic primitives used in this thesis context is provided 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 [23]. This will be important in the algorithms anal-

Algorithm 1 Michael and Scott's Lock-Free Queue

```
function INITIALIZE(Q : pointer to queue_t)
    node = new node()                                    ▷ Allocate a dummy node
    node.next.ptr = NULL                                 ▷ Make it the only node in the list
    Q.Head = node                                        ▷ Both Head and Tail point
    Q.Tail = node                                        ▷ to this dummy node
end function
function ENQUEUE(Q : pointer to queue_t, value : data_type)
    node = new node()                                    ▷ Allocate a new node from the free list
    node.value = value                                   ▷ Copy enqueue value into node
    node.next.ptr = NULL                                 ▷ Set next pointer of node to NULL
    loop                                                 ▷ Keep trying until Enqueue is done
        tail = Q.Tail                                    ▷ Read Tail (pointer + count) together
        next = tail.ptr.next                             ▷ Read next ptr + count together
        if tail == Q.Tail then                           ▷ Are tail & next consistent?
            if next.ptr == NULL then                     ▷ Tail is the last node?
                if CAS(&tail.ptr.next, next, ⟨node, next.count + 1⟩) then
                    break                                ▷ Link the new node; Enqueue is done
                end if
            else                                         ▷ Tail not pointing to the last node
                CAS(&Q.Tail, tail, ⟨ next.ptr, tail.count+1⟩)   ▷ Move Tail forward (helping another enqueuer)
            end if
        end if
    end loop
    CAS(&Q.Tail, tail, ⟨node, tail.count + 1⟩)            ▷ Final attempt to swing Tail to the inserted node
end function
function DEQUEUE(Q : pointer to queue_t, pvalue : pointer to data_type)
    loop                                                 ▷ Keep trying until Dequeue is done
        head = Q.Head
        tail = Q.Tail
        next = head.ptr.next                             ▷ Read head->next
        if head == Q.Head then                           ▷ Still consistent?
            if head.ptr == tail.ptr then                 ▷ Empty or Tail behind?
                if next.ptr == NULL then                 ▷ Queue is empty
                    return FALSE
                else                                     ▷ Tail is behind, help move it
                    CAS(&Q.Tail, tail, ⟨next.ptr, tail.count + 1⟩)
                end if
            else                                         ▷ No need to adjust Tail
                *pvalue = next.ptr.value                 ▷ Read value before CAS
                if CAS(&Q.Head, head, ⟨next.ptr, head.count + 1⟩) then
                    break                                ▷ Dequeue is done
                end if
            end if
        end if
    end loop
    free(head.ptr)                                       ▷ Safe to free old dummy node
    return TRUE
end function
```

[22]

ysed later in this thesis, since these primitives are used to implement wait-free synchronisation. There are different kinds 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 architectures, is 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" [24]
- "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" [24]

[24]

Compare and Swap (CAS)

In addition 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" [24].

- "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." [24]

The problem with CAS, beyond 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 fundamental limitation of CAS. One solution would be to replace CAS with LL/SC, but that is not possible on x86 processors. Therefore, other solutions are needed, which are discussed in chapter 6. [24]

Double-Width Compare and Swap (DWCAS)

This is a CAS on two neighbouring memory locations. [24]

Double Compare and Swap (DCAS)

Sometimes also called CAS2, is a CAS on two independent memory locations. [24]

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 the value v in location R and returns the previous value stored in R. This operation always succeeds. [25].

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." [24]

Fetch and Store (FAS)

This atomically stores a value in 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 than CAS, if conditions before updating do not need to be checked. [26]

2.5. Wait-Free Synchronisation

Lock-freedom solves the problem of a system becoming stuck in a deadlock. However, this is not enough. For example, in a fully automated car, it is undesirable for any process to fail to complete its task, as this could mean that some processes responsible for braking would not finish their work in a worst-case scenario. And in such an occasion where the car would need to brake, a fatal collision would be the outcome. Consequently, a solution is necessary where every process completes its task in a finite number of steps, rather than just one process. So, something is needed that builds upon such mechanisms and extends them. This is exactly what wait-free synchronisation 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, no longer possible. Additionally, priority inversion is eliminated because processes no longer have to wait for other processes to complete. This ensures system responsiveness and predictability, thereby enabling the definition of strict timing constraints required for HRTS applications. But even wait-free algorithms introduce one problem. Wait-free algorithms are, in most cases, slower than their lock-free counterparts in execution. A solution to this will be addressed in chapter 3 and analysed in more depth in chapter 5.

2.6. Rust Programming Language

The question now is which programming language suits best for this kind of algorithms. Since fast communication between processes is crucial to meet all HRTS timing constraints, the C programming language is a suitable 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 collector, which gets active and stops all processes from working to clean up allocated but no longer used memory space. Because of that, all RTOS are written in C. The primary issue with C is that it lacks memory safety, as it implements memory operations that are prone to buffer overflows or control-flow attacks. In the industry, around 70% of vulnerabilities happen due to memory safety issues. If the real-time application were to run on an isolated system with no internet connection, this would not be a problem. However, in modern automation, where systems must be connected to the internet for data exchange, such systems are prone to security attacks. RTS is nowadays an integral part of various connected devices, including critical fields such as health and transportation. Consequently, the security of such devices must be ensured. With the Rust programming language, the problems of memory safety are gone. The difference to C is that it can be as fast as C, while also supporting low-level control and high-level programming features, and providing memory safety features. 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 programme has guaranteed memory safety. In the model, every value has a single owner represented by a variable. The owner is responsible for 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 called borrowing. This allows the usage of these values without transferring the ownership. These references have their lifetime, which can be explicitly defined by the programmer or implicitly inferred by Rust's compiler. This ensures that the references are valid and do not exist longer than needed. Hence, this can play a role in lock-freedom, which is required for wait-free synchronisation, since shared resources can be shared with this ownership concept. Additionally, Rust is a type-safe language, which can be helpful during implementation to avoid bugs and errors. As seen, Rust is a good choice for implementing wait-free synchronisation mechanisms for IPC in RTS. [9], [27].

Further mechanisms on how Rust addresses various common memory safety issues will not be discussed in detail, as that would exceed the scope of this thesis. It is essential to understand the basics of how Rust is a type-safe and memory-safe programming language to comprehend why it is used for this work.

3. Related Work

The early foundations regarding wait-freedom were laid indirectly by Leslie Lamport in 1983 [28]. While his work did not directly address or formally define wait-freedom or lock-freedom, it laid the groundwork for Maurice Herlihy to define wait-freedom [1].
In 1983, Leslie Lamport introduced a formal method for writing and proving the correctness of any concurrent module in a simple and modular way, independent of the data structure used, which he referred 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). In contrast, liveness requirements define what must eventually happen (e.g., a non-empty queue must eventually allow dequeueing to occur). He also defines the usage of action sets and environment constraints, which separate the module action from the environments (e.g., the program in which the data structure runs). 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. [28]

Building on methodologies 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 [29]. This correctness condition was then used to formalise 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 the following three constraints:

- Linearisability: 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' execution (for later understanding: 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).

[1], [29]

Herlihy's universal construction and principles (or work that builds upon his work) appear conceptually throughout all of these wait-free algorithms [4], [25], [26], [30]–[46]. [1]

Additionally, in 1983, Lamport presented a concurrent lock-free FIFO queue implementation using a circular buffer. It works by using an array Q with two pointers, HEAD and TAIL, where elements are added at TAIL and removed at HEAD. The producer first checks if the queue is full by testing if TAIL - HEAD equals the queue capacity m, and if so, it busy-waits. When space is available, it transfers the element bit-by-bit into Q at position TAIL modulo m, then atomically increments TAIL to commit the addition. Similarly, the consumer checks if the queue is empty by testing if TAIL - HEAD equals 0, and if so, it busy-waits. When an element is available, it extracts the element bit-by-bit from Q at position HEAD modulo m using shift operations, then atomically increments HEAD to commit the removal. This means that while data is being shifted, other operations can observe partial states, but the queue still maintains its FIFO properties correctly because an element is only considered "in" the queue after TAIL is incremented and only considered "removed" after HEAD is incremented. Even though busy-waits are used, the queue will eventually fill and be dequeued, so the producer and consumer will finish in a finite number of steps, making the Lamport queue wait-free, even though that term was not defined at the time. [28]

The Lamport queue is the basis of many more wait-free queues later invented, discussed in chapter 5.

Kogan and Petrank later invented a method called fast-path slow-path, where, first, a lock-free method (the fast-path) is typically used to attempt to complete an operation, as lock-free algorithms are generally faster than wait-free algorithms. A maximum number of steps bounds these lock-free paths, 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). Therefore, in cases where the fast path succeeds frequently without switching to the slow path, the algorithm generally completes in a shorter time than 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 in chapter 5. [4]

4. Methodology

To achieve the goal defined in section 1.2, first, all wait-free data structures that can be used for IPC through shared memory in HRTS need to be identified. To identify all existing wait-free data structures, a method was employed that is more commonly used in mapping studies or literature reviews. Multiple Python scripts were implemented to do this, as seen in the accompanying GitHub repository [47]. A web scraper script was written to scrape over Google Scholar with the following queries:

- “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, a list of 1324 papers was found. The papers were then recorded in a CSV file split into query, rank (number of paper), title, year, authors, venue, citations, abstract snippet, full_abstract and url with “;” as a delimiter. To extract all of this, the information in Google Scholar was extracted, and then, for the full abstract, the scraper went to 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 inaccessible, the whole paper was written instead into that cell. Because a lot of scientific web pages will put captchas if continuous requests are made, “undetected_chromedriver” was imported and used as the web driver. It is an enhanced version of ChromeDriver which bypasses anti-bot detections. After that, a regex analyser was implemented to analyse the abstracts of the found papers on the words “lock-free”, “wait-free”, and “obstruction-free” and also again without a hyphen in between these words. If these keywords were not found in the abstract, the paper was removed from the CSV. The abstracts of papers with the tag “ABSTRACT_NOT_FOUND” had to be analysed manually. This left 475 papers that contained at least one of these words in their abstract. After that, the duplicates were removed by the algorithm, resulting in 325 remaining papers. The duplicates were removed by checking the URL of the paper. Now what was left had to be manually analysed to see if the paper was relevant to the topic. Since LibreOffice and Microsoft Excel have a limit of 32,767 characters per cell, an abstract splitter was built to split the abstract into multiple cells. Analysis was conducted by reading the paper and verifying whether it presented a wait-free FIFO queue. The reason why only FIFO queues were considered will be explained in the next chapter. While doing that, a backwards and forward search was also done to find more papers. Only papers were considered if the queue described in the paper met the following criteria:

- All three of Herlihy's constraints were met, which are listed in chapter 3.
- The algorithms can be implemented on the x86 architecture using Rust, as this is the architecture and programming language 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 queues were identified through web scraping, and 3 queues were discovered through a backwards and forward search of the included papers. The queues were then split into 4 contention categories:

- MPMC queues with 6 queues [30], [32], [34]–[37]
- MPSC queues with 4 queues [26], [38], [40], [41]
- SPMC queues with 1 queue [25]
- SPSC queues with 11 queues [28], [41], [42], [44]–[46]

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 data structure.

5. Analysing existing Wait-Free Data Structures and Algorithms

After finding and identifying existing wait-free algorithms, they need to be analysed as stated in section 1.2 to meet the primary objective of this work. That is exactly what this chapter will be about.

5.1. Optimal Wait-Free Data Structure

Before analysing the algorithms, first, it must be determined which data structure to use for the implementation of wait-free synchronisation for IPC, since only paper were cosidered describing FIFO data structures. M. Herlihy showed that every sequential data structure can be made wait-free [1]. Therefore, it is important to choose the optimal data structure for our use. Considering that the reason for this work is to optimise modern manufacturing and automation, some form of correct data flow order is also necessary for correct workflow, for instance, in a modern manufacturing line or more critically in a driverless car. Hence, FIFO queues are a natural fit. This is 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 preserved without the need for 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 for additional functions to just get rid of undesired side effects. Furthermore, in a queue only two operations exist: an enqueue and a dequeue operation. All the other data structures introduce more operations and therefore more complexity and therefore more performance overhead. The fewer operations that 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 being used, limiting this thesis to queues is reasonable. [40]

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. The choice of algorithm will be based on contention. Since all of them have different complexities 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). Therefore, 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 benchmark (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 the 4 different categories are necessary. The reason for this is that, for instance, the best-performed MPMC algorithm could outperform all other algorithms even for their contention category, considering that an 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 an MPMC use case could have extreme overhead for an SPSC case. The implementation will be discussed in chapter 6, 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. The subsections will also shortly describe how the enqueue and dequeue operations 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 chapter 6, it will be elaborated on 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 [47]. These are not detailed here as such explanations would provide limited value to the topic of this work, and whilst 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 simplest form of IPC. In SPSC, there is nearly no contention from other processes, because only one producer and one consumer are working. The only contention is between the consumer and the producer. This leads to the producer and consumer finishing in a bounded number of steps without special synchronisation techniques like helping or atomic primitives. The only concern is that the data the consumer reads is consistent. Different approaches were tested in different papers, which 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 synchronisation, based on the algorithm originally proposed by Leslie Lamport in 1983 [28] and shown here in algorithm 2 following Maffione et al.'s version [45]. 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 behaviour 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 behaviour creates the circular buffer structure, allowing the fixed-size array to be reused continuously. It can be observed that the consumer and producer just carry out a finite number of operations without waiting for any condition from the other process, which leads to them finishing in a finite number of steps. 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 U. Drepper [48], cache misses occur when different processes access variables that reside on the same cache line. A cache miss is when the requested data is not in the local CPU core's cache and must be fetched from another core's cache or the main memory, with the cache coherence protocol ensuring memory consistency across all cores. 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.

Lazy Lamport Queue (LLQ)

Reduces the described cache misses by postponing index reads until necessary, as shown in algorithm 3. In addition to Lamport's 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. Moreover, 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. This queue only adds a bounded number of additional reads on top of the Lamport queue, still making the design wait-free. [45]

Algorithm 2 Lamport's Queue [45]

```
function LQ_ENQUEUE(q, e)
    if q.write − q.read = N then                     ▷ Check if full
        return −1                                    ▷ No space
    end if
    q.slots[q.write ∧ q.mask] ← e
    store_release_barrier()
    q.write ← q.write + 1
    return 0
end function

function LQ_DEQUEUE(q)
    if q.read = q.write then                         ▷ Check if empty
        return NULL_ELEM                             ▷ Queue empty
    end if
    load_acquire_barrier()
    e ← q.slots[q.read ∧ q.mask]
    q.read ← q.read + 1
    return e
end function
```

[45]

Algorithm 3 LLQ Operations [45]

```
function LLQ_ENQUEUE(q, e)
    if q.write − q.read_shadow = N − K then          ▷ Lazy load check
        q.read_shadow ← q.read                       ▷ Update shadow
        if q.write − q.read_shadow = N − K then
            return −1                                ▷ No space
        end if
    end if
    q.slots[q.write ∧ q.mask] ← e
    store_release_barrier()
    q.write ← q.write + 1
    return 0
end function

function LLQ_DEQUEUE(q)
    if q.read = q.write_shadow then                  ▷ Lazy load check
        q.write_shadow ← q.write                     ▷ Update shadow
        if q.read = q.write_shadow then
            return NULL_ELEM
        end if
    end if
    load_acquire_barrier()
    e ← q.slots[q.read ∧ q.mask]
    q.read ← q.read + 1
    return e
end function
```

Batched Lamport Queue (BLQ)

Extends LLQ with explicit batching to further reduce synchronisation 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 $B = 32$, synchronisation overhead is amortised 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. The added batching logic on top of Lamport's queue only adds a bounded amount of additional writes, not changing the overall bounded step logic of Lamport's queue. [45]

Algorithm 4 BLQ Operations [45]

```
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)

Synchronisation 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, FFQ's 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. Otherwise, the queue is full and the operation returns. The consumer follows a similar pattern by reading from the current slot position in line 11 and then checking in line 12 if 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 is how the synchronisation happens and why the producer and consumer end in a finite number of steps. Neither consumer nor producer will end up in a retry loop waiting for the other. Additionally, the shared memory access was reduced from three (`head`, `tail`, `buffer`) to just one (`buffer` slot). Each thread maintains its own private index that never needs synchronisation. The producer tracks where to write next through `head`, whilst the consumer tracks where to read next through `tail`. The `NULL` value serves a dual purpose as both an empty indicator and the synchronisation mechanism. Whilst 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. It can be seen that FFQ's enqueue and dequeue function do complete in exactly 3 steps without any loops or retry mechanisms, leading to a wait-free trait. [46]

Algorithm 5 FFQ Operations [46]

```
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 of FFQ 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, the producer and consumer operate on different cache lines, reducing cache misses even more. Wait-freedom is preserved because the look-ahead check adds only one bounded operation every H items on top of FFQ, and the delayed clearing in `iffq_dequeue_publish` executes a loop bounded by algorithm parameters, maintaining the original bounded step guarantee of FFQ.[45]

Algorithm 6 IFFQ Operations [45]

```
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
```

Batched Improved FastForward Queue (BIFFQ)

Gets rid of IFFQ's 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 behaviour 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. Whilst theoretical worst-case behaviour is similar to IFFQ, practical measurements show significant improvement when the queue operates near empty, making BIFFQ effective across all operating conditions. Like BLQ, the buffering mechanism maintains wait-freedom through bounded local operations and a publish loop limited by buffer size on top of FFQ's bounded steps, ensuring every operation completes within a fixed number of steps regardless of the consumer's behaviour. [45]

Algorithm 7 BIFFQ Operations [45]

```
function BIFFQ_WSPACE(q, needed)
    space ← q.limit − q.write
    if space < needed then
        return space                              ▷ Force limit update
    end if
    return space
end function

function BIFFQ_ENQUEUE_LOCAL(q, e)
    q.buf[q.buffered] ← e                         ▷ Store in buffer
    q.buffered ← q.buffered + 1
end function

function BIFFQ_ENQUEUE_PUBLISH(q)
    for i ← 0 to q.buffered − 1 do
        q.slots[q.write ∧ q.mask] ← q.buf[i]      ▷ Fast burst
        q.write ← q.write + 1
    end for
    q.buffered ← 0
    q.limit ← q.write + H                         ▷ Update limit
end function
```

B-Queue

Enhances the performance of 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 lines 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 whilst maintaining cache line separation and preventing deadlocks. It can be observed that the producer is just executing a constant number of operations. The consumer's backtracking loop is bounded by at most $\log_2$(`BATCH_SIZE`) iterations since `batch_size` halves each time as seen in line 40, which guarantees that this loop will end in a finite number of steps. [44]

Algorithm 8 B-Queue with Self-Adaptive Backtracking [44]

```
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
```

Dynamic 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 the queue size cannot be predetermined. The implementation maintains a dummy head node `head` to ensure the 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 minimise 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 the 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. As one can see, the consumer and also the producer do not have any kind of loops and just carry out a small set of operations, which leads to the wait-freedom of both. The node caching is for improving performance, because reading and referencing the pointer so often causes memory accesses spread over multiple cache lines. As shown earlier, this leads to cache misses. [42]

Algorithm 9 dSPSC Operations [42]

```
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
```

Unbounded Single Producer Single Consumer (uSPSC)

An unbounded queue that links multiple already wait-free Lamport Queues to combine the cache efficiency of Lamport's 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 whilst 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 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 behaviour whilst providing unbounded capacity. The logic built on top of the wait-free Lamport queues as seen does still maintain wait-freedom, since the synchronisation is happening inside each individual Lamport queue, whilst neither the consumer nor producer gets stuck in a loop waiting for each other. [42]

Algorithm 10 uSPSC Operations[42]

```
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
```

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 backwards 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. Whilst adding an extra copy per element from batch to buffer, the improved cache behaviour from reduced traffic from the coherence protocol compensates for this overhead. As seen, this queue just adds extra logic on top of the Lamport queue, making it still wait-free, since `BATCH_SIZE` is a constant and the loops in the queue are therefore bounded. [42]

Algorithm 11 mSPSC Operations[42]

```
 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 Petrovic Queue (JPQ) (SPSC variant)

A queue specifically for composability in larger MPSC structures, as shown in algorithm 12. Unlike traditional SPSC queues, this implementation includes `readFront` operations that enable 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, signalling 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 whilst 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. Wait-freedom is achieved because there are no loops in the enqueue and dequeue functions. [41]

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, whilst a single consumer dequeues items. This includes implementing other strategies, 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 Petrovic Queue (JPQ) (MPSC variant)

Achieves logarithmic time complexity by distributing the global queue across n local SPSC queues implemented like in section 5.2.1 and organised 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 the 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,

Algorithm 12 JPQ (SPSC variant) Operations [41]

```
function ENQUEUE(q, v)
    newNode ← new Node()                              ▷ Create new dummy node
    tmp ← q.Last                                      ▷ Get current dummy tail
    tmp.val ← v                                       ▷ Convert dummy to data node
    tmp.next ← newNode                                ▷ Link new dummy
    q.Last ← newNode                                  ▷ Update tail pointer
end function

function DEQUEUE(q)
    tmp ← q.First                                     ▷ Get head node
    if tmp = q.Last then                              ▷ Only dummy remains?
        return ⊥                                      ▷ Queue empty
    end if
    retval ← tmp.val                                  ▷ Read value
    q.Help ← retval                                   ▷ Help concurrent readFront
    q.First ← tmp.next                                ▷ Remove from queue
    if tmp = q.Announce then                          ▷ Was announced by readFront?
        tmp′ ← q.FreeLater                            ▷ Get old deferred node
        q.FreeLater ← tmp                             ▷ Defer current node
        free(tmp′)                                    ▷ Free old deferred node
    else
        free(tmp)                                     ▷ Free immediately
    end if
    return retval
end function

function READFRONT_E(q)                               ▷ Called by enqueuer
    tmp ← q.First                                     ▷ Read head pointer
    if tmp = q.Last then                              ▷ Queue empty?
        return ⊥
    end if
    q.Announce ← tmp                                  ▷ Announce to prevent free
    if tmp ≠ q.First then                             ▷ Head changed (was dequeued)?
        retval ← q.Help                               ▷ Use helped value
    else
        retval ← tmp.val                              ▷ Read directly
    end if
    return retval
end function

function READFRONT_D(q)                               ▷ Called by dequeuer
    tmp ← q.First
    if tmp = q.Last then
        return ⊥
    end if
    return tmp.val                                    ▷ Safe - no concurrent dequeue
end function
```

another process updates the node. If the second refresh also fails, that process must have read the updated 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, whilst the space complexity remains $O(n+m)$ where m is the number of queued items. The wait freedom is achieved by letting the producers work only in their own local queue, avoiding any contention with other producers. The dequeue is just a set of instructions without loops. [41]

Algorithm 13 JPQ (MPSC variant) Operations [41]

```
function ENQUEUE(q, p, v)
    tok ← LL(q.counter)                                  ▷ Read timestamp
    SC(q.counter, tok + 1)                               ▷ Try increment
    enqueue2(q.Q[p], (v, (tok, p)))                      ▷ Add timestamp to local queue
    PROPAGATE(q, q.Q[p])
end function

function DEQUEUE(q, p)
    [t, id] ← read(q.T.root)                             ▷ Get min producer
    if id = ⊥ then
        return ⊥
    end if
    ret ← dequeue2(q.Q[id])
    PROPAGATE(q, q.Q[id])
    return ret.val
end function

function PROPAGATE(q, localQueue)
    currentNode ← localQueue
    repeat
        currentNode ← parent(currentNode)
        if ¬REFRESH(q, currentNode) then                 ▷ First try
            REFRESH(q, currentNode)                      ▷ Second ensures correctness
        end if
    until currentNode = q.T.root
end function

function REFRESH(q, node)
    LL(node)                                             ▷ Load-link node
    stamps ← read timestamps from node's children
    minT ← minimum timestamp from stamps
    return SC(node, minT)                                ▷ Store-conditional
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 wait-freedom is achieved by the single atomic FAS for producers. [26]

Algorithm 14 Drescher's Wait-Free MPSC Queue Operations

```
dummy.next ← 0
head ← &dummy
tail ← &dummy

procedure ENQUEUE(guard, item)
    item.next ← 0                                   ▷ Clear next pointer
    prev ← FAS(guard.tail, item)                    ▷ Atomic swap tail
    prev.next ← item                                ▷ Link to new item
end procedure

function DEQUEUE(guard)
    item ← guard.head
    next ← guard.head.next
    if next = 0 then                                ▷ Empty queue?
        return ⊥
    end if
    guard.head ← next
    if item = &dummy then                           ▷ Dequeued dummy?
        ENQUEUE(guard, item)                        ▷ Re-enqueue dummy
        if guard.head.next = 0 then                 ▷ Still empty?
            return ⊥
        end if
        guard.head ← guard.head.next
        return next
    end if
    return item
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 the `DEQUEUE` function in algorithm 16. Unlike other linked-list queues that allocate nodes per element, Jiffy amortises 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 synchronisation 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` (uninitialised), `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 linearisability when producers stall: if the head element is still `empty`, the consumer scans forward to find a `set` element in line 20, then rescans backwards 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 too much memory even with stalled producers. This achieves wait-free progress guarantees with minimal synchronisation. Producers only need one FAA per enqueue, whilst the consumer performs no atomic operations at all, skipping still unfinished enqueues. [40]

Algorithm 15 Jiffy MPSC Queue Enqueue Operation [40]

```
function ENQUEUE(data)
    location ← FAA(tail, 1)                                  ▷ Reserve global index
    tempTail ← tailOfQueue
    while location is in unallocated buffer do               ▷ Beyond last buffer?
        if tempTail.next = NULL then
            newArr ← new BufferList()
            if CAS(tempTail.next, NULL, newArr) then
                CAS(tailOfQueue, tempTail, newArr)
            else
                delete newArr                                ▷ Another thread succeeded
            end if
        end if
        tempTail ← tailOfQueue                               ▷ Move to new buffer
    end while
    while location not in tempTail's buffer do               ▷ Location in earlier buffer?
        tempTail ← tempTail.prev                             ▷ Walk backward
    end while
    index ← location − tempTail.startIndex                   ▷ Buffer-local index
    tempTail.buffer[index].data ← data
    tempTail.buffer[index].isSet ← SET                       ▷ Mark as ready
    if index = 1 AND tempTail is last buffer then            ▷ Second slot?
        newArr ← new BufferList()                            ▷ Proactive allocation
        if NOT CAS(tempTail.next, NULL, newArr) then
            delete newArr
        end if
    end if
end function
```

DQueue

Combines local buffering with a segmented shared queue to minimise synchronisation overhead, as shown in function `ENQUEUE(data)` in algorithm 17 and in the `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 synchronisation, 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

Algorithm 16 Jiffy MPSC Queue Dequeue Operation [40]

```
function DEQUEUE
    n ← headOfQueue.buffer[head]
    while n.isSet = HANDLED do                                  ▷ Skip dequeued items
        head ← head + 1
        if end of buffer then
            move to next buffer and delete current
        end if
        n ← headOfQueue.buffer[head]
    end while
    if queue is empty then
        return ⊥
    end if
    if n.isSet = SET then                                       ▷ Ready to dequeue?
        data ← n.data
        n.isSet ← HANDLED
        head ← head + 1
        return data
    end if
    if n.isSet = EMPTY then                                     ▷ Incomplete enqueue?
        tempN ← Scan(find first SET element)
        if no SET element found then
            return ⊥
        end if
        Rescan(n, tempN)                                        ▷ Check for newly set elements
        data ← tempN.data
        tempN.isSet ← HANDLED
        return data
    end if
end function

function SCAN                                                   ▷ Find first SET element
    for each element from current position do
        if element.isSet = SET then
            return element
        end if
        if entire buffer is HANDLED then
            fold queue (delete buffer)
        end if
    end for
    return NULL
end function

function RESCAN(start, end)                                     ▷ Check for ordering violations
    for each element from start to end do
        if element.isSet = SET then
            end ← element                                       ▷ Found earlier SET element
            restart scan from start
        end if
    end for
end function
```

allocates new segments on-demand using CAS in line 29. To maintain wait-freedom when 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 synchronisation 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 linearisation point at line 14, where it increments the head pointer. [38]

Algorithm 17 DQueue MPSC Queue Enqueue Operation [38]

```
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 synchronised to read consistently without any unwanted behaviour. Multiple producers were simpler, since making producers write specific data for each is not so hard. In this case every consumer has

Algorithm 18 DQueue MPSC Queue Dequeue Operation [38]

```
function DEQUEUE(Consumer c)
    seg ← find_segment(c.cseg, q.head)
    if c.cseg ≠ seg then
        c.cseg ← seg                                   ▷ Update segment cache
    end if
    cell ← seg.cell[q.head mod N]
    if cell = ⊥ then                                   ▷ Empty cell?
        if q.head = q.tail then
            return EMPTY
        else
            help_enqueue()                             ▷ Help stalled producers
        end if
    end if
    q.head ← q.head + 1                                ▷ Linearisation point
    return cell
end function

function HELP_ENQUEUE
    for each Producer p in system do
        for each Request r in p.local_buffer do
            pos ← r.cid
            val ← r.val
            seg ← find_segment(p.pseg, pos)
            if seg.id > pos/N then
                break                                  ▷ Producer moved past
            end if
            if seg.cell[pos mod N] = ⊥ then
                seg.cell[pos mod N] ← val              ▷ Help write
            end if
        end for
    end for
end function
```

to be synchronised so that when one item is consumed by a consumer, it cannot be consumed again from another and also the ordering has to be kept. That is most probably also the reason why 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 producer maintains two persistent local variables, an `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 ⊤ into the reserved cell in line 24, retrieving either the enqueued value or ⊥ (empty). The use of swap instead of plain registers is important because it allows the producer to detect consumer interference (by finding ⊤) 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), whilst consumers always complete in exactly 3 steps. [25]

Algorithm 19 David's Queue Operations [25]

```
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 synchronising 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 Petrank's 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 at most n other slower threads. 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 utilise 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 the list in line 15, then clearing the pending flag in line 35 of HELP_FINISH_ENQ, and finally updating 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 the 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. So after a thread helps at most n other threads, the thread's own operation has been completed by a helper, or all n threads are now helping this thread, guaranteeing it completes. This achieves wait-free progress with $O(n)$ steps per operation, where n is the number of threads, leading to an $O(n^2)$ bound. [30]

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 requests 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 consumers 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

Algorithm 20 Kogan and Petrank's Queue Enqueue Operation [30]

```
function ENQUEUE(value)
    phase ← MAXPHASE +1                                        ▷ Choose phase
    state[tid] ← OpDesc(phase, true, true, Node(value, tid))
    HELP(phase)                                                ▷ Help all ops ≤ phase
    HELP_FINISH_ENQ                                            ▷ Ensure tail updated
end function

procedure HELP_ENQ(tid, phase)
    while ISSTILLPENDING(tid, phase) do
        last ← tail
        next ← last.next
        if last = tail then                                    ▷ Validate read
            if next = null then                                ▷ Can append?
                if ISSTILLPENDING(tid, phase) then
                    if CAS(last.next, null, state[tid].node) then
                        HELP_FINISH_ENQ
                        return
                    end if
                end if
            else                                               ▷ Help pending enqueue
                HELP_FINISH_ENQ
            end if
        end if
    end while
end procedure

procedure HELP_FINISH_ENQ
    last ← tail
    next ← last.next
    if next ≠ null then
        tid ← next.enqTid                                      ▷ Thread that owns node
        desc ← state[tid]
        if last = tail and state[tid].node = next then
            newDesc ← OpDesc(state[tid].phase, false, true, next)
            CAS(state[tid], desc, newDesc)                     ▷ Clear pending
            CAS(tail, last, next)                              ▷ Update tail
        end if
    end if
end procedure

function MAXPHASE
    max ← −1
    for i ← 0 to NUM_THREADS − 1 do
        phase ← state[i].phase
        if phase > max then
            max ← phase
        end if
    end for
    return max
end function
```

`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 is permanent and indicates ownership. The dequeue handles empty queues through a "give-

Algorithm 21 Kogan and Petrank's Queue Dequeue Operation [30]

```
function DEQUEUE
    phase ← MAXPHASE +1
    state[tid] ← OpDesc(phase, true, false, null)
    HELP(phase)
    HELP_FINISH_DEQ
    node ← state[tid].node
    if node = null then
        throw EmptyException
    end if
    return node.next.value
end function

procedure HELP_DEQ(tid, phase)
    while ISSTILLPENDING(tid, phase) do
        first ← head
        last ← tail
        next ← first.next
        if first = head then                                    ▷ Validate read
            if first = last then                                ▷ Queue might be empty
                if next = null then                             ▷ Confirmed empty
                    desc ← state[tid]
                    if last = tail and ISSTILLPENDING(tid, phase) then
                        newDesc ← OpDesc(state[tid].phase, false, false, null)
                        CAS(state[tid], desc, newDesc)
                    end if
                else
                    HELP_FINISH_ENQ                             ▷ Help enqueue
                end if
            else                                                ▷ Queue not empty
                desc ← state[tid]
                node ← desc.node
                if not ISSTILLPENDING(tid, phase) then
                    break
                end if
                if first = head and node ≠ first then
                    newDesc ← OpDesc(state[tid].phase, true, false, first)
                    if not CAS(state[tid], desc, newDesc) then
                        continue
                    end if
                end if
                CAS(first.deqTid, −1, tid)                      ▷ Lock node
                HELP_FINISH_DEQ
            end if
        end if
    end while
end procedure
```

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 roll back their request in line 13 but must ensure no concurrent thread assigns them a node. This 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

Algorithm 22 Kogan and Petrank's Queue Dequeue Helping Operations [30]

```
procedure HELP_FINISH_DEQ
    first ← head
    next ← first.next
    tid ← first.deqTid                                          ▷ Thread that locked node
    if tid ≠ −1 then
        desc ← state[tid]
        if first = head and next ≠ null then
            newDesc ← OpDesc(state[tid].phase, false, false, state[tid].node)
            CAS(state[tid], desc, newDesc)                      ▷ Clear pending
            CAS(head, first, next)                              ▷ Advance head
        end if
    end if
end procedure

procedure HELP(phase)
    for i ← 0 to NUM_THREADS − 1 do
        desc ← state[i]
        if desc.pending and desc.phase ≤ phase then
            if desc.enqueue then
                HELP_ENQ(i, phase)
            else
                HELP_DEQ(i, phase)
            end if
        end if
    end for
end procedure
```

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 node itself), achieving minimal memory overhead of $O(N_{threads})$ compared to the $O(N^2_{threads})$ of other wait-free queues. [34]

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, whilst 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, whilst consumers ensure values are available to dequeue.

Algorithm 23 Turn Queue Enqueue Operation [34]

```
function ENQUEUE(item)
    if item = null then throw InvalidArgument
    end if
    myIdx ← GETINDEX
    myNode ← new Node(item, myIdx)
    enqueuers[myIdx] ← myNode
    for i ← 0 to maxThreads − 1 do
        if enqueuers[myIdx] = null then                          ▷ Request completed
            hp.CLEAR
            return
        end if
        ltail ← hp.PROTECTPTR(kHpTail, tail)
        if ltail ≠ tail then continue
        end if
        if enqueuers[ltail.enqTid] = ltail then                  ▷ Clear old request
            tmp ← ltail
            CAS(enqueuers[ltail.enqTid], tmp, null)
        end if
        for j ← 1 to maxThreads do                               ▷ Find next request
            nodeHelp ← enqueuers[(j + ltail.enqTid) mod maxThreads]
            if nodeHelp ≠ null then
                nullnode ← null
                CAS(ltail.next, nullnode, nodeHelp)              ▷ Try append
                break
            end if
        end for
        lnext ← ltail.next
        if lnext ≠ null then                                     ▷ Advance tail if needed
            CAS(tail, ltail, lnext)
        end if
    end for
    enqueuers[myIdx] ← null                                      ▷ Cleanup if not helped
    hp.CLEAR
end function
```

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 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 linearisability through careful ordering. Enqueues linearise when `T` moves past their cell index, whilst dequeues linearise 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. The slow-path is always entered after at most a thread tried the fast-path `PATIENCE` times (line 2), which is a defined constant. [37]

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

Algorithm 24 Turn Queue Dequeue Operation [34]

```
function DEQUEUE
    myIdx ← GETINDEX
    prReq ← deqself[myIdx]                                   ▷ Save previous request
    myReq ← deqhelp[myIdx]
    deqself[myIdx] ← myReq                                   ▷ Open request
    for i ← 0 to maxThreads − 1 do
        if deqhelp[myIdx] ≠ myReq then break                 ▷ Request satisfied
        end if
        lhead ← hp.PROTECTPTR(kHpHead, head)
        if lhead ≠ head then continue
        end if
        if lhead = tail then                                 ▷ Queue empty
            deqself[myIdx] ← prReq                           ▷ Rollback
            GIVEUP(myReq, myIdx)
            if deqhelp[myIdx] ≠ myReq then                   ▷ Check if helped
                deqself[myIdx] ← myReq
                break
            end if
            hp.CLEAR
            return null
        end if
        lnext ← hp.PROTECTPTR(kHpNext, lhead.next)
        if lhead ≠ head then continue
        end if
        if SEARCHNEXT(lhead, lnext) ≠ NOIDX then
            CASDEQANDHEAD(lhead, lnext, myIdx)
        end if
    end for
    myNode ← deqhelp[myIdx]
    lhead ← hp.PROTECTPTR(kHpHead, head)
    if lhead = head and myNode = lhead.next then
        CAS(head, lhead, myNode)                             ▷ Help advance head
    end if
    hp.CLEAR
    hp.RETIRE(prReq)                                         ▷ Retire previous node
    return myNode.item
end function
```

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 `seqid mod capacity` in line 8. In the common case, threads replace an `EmptyNode` with a 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 a `EmptyNode` with sequence ID incremented by the buffer capacity in line 9. The dequeue succeeds when replacing an `ElemNode` with a matching sequence ID in line 42 via CAS. If encountering an `EmptyNode`

Algorithm 25 Turn Queue Dequeue Helper Functions [34]

```
function SearchNext(lhead, lnext)
    turn ← lhead.deqTid
    for idx ← turn + 1 to turn + maxThreads do
        idDeq ← idx mod maxThreads
        if deqself[idDeq] ≠ deqhelp[idDeq] then continue
        end if
        if lnext.deqTid = NOIDX then
            CAS(lnext.deqTid, NOIDX, idDeq)                    ▷ Assign node
        end if
        break
    end for
    return lnext.deqTid
end function

procedure CasDeqAndHead(lhead, lnext, myIdx)
    ldeqTid ← lnext.deqTid
    if ldeqTid = myIdx then                                    ▷ My node
        deqhelp[ldeqTid] ← lnext                               ▷ Close request
    else
        ldeqhelp ← hp.ProtectPtr(kHpDeq, deqhelp[ldeqTid])
        if ldeqhelp ≠ lnext and lhead = head then
            CAS(deqhelp[ldeqTid], ldeqhelp, lnext)             ▷ Help close
        end if
    end if
    CAS(head, lhead, lnext)                                    ▷ Advance head
end procedure

procedure GiveUp(myReq, myIdx)
    lhead ← head
    if deqhelp[myIdx] ≠ myReq then return                      ▷ Already helped
    end if
    if lhead = tail then return                                ▷ Still empty
    end if
    hp.ProtectPtr(kHpHead, lhead)
    if lhead ≠ head then return
    end if
    lnext ← hp.ProtectPtr(kHpNext, lhead.next)
    if lhead ≠ head then return
    end if
    if SearchNext(lhead, lnext) = NOIDX then
        CAS(lnext.deqTid, NOIDX, myIdx)                        ▷ Self-assign
    end if
    CasDeqAndHead(lhead, lnext, myIdx)
end procedure
```

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 operation 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

Algorithm 26 YMC Queue Enqueue Operation [37]

```
function ENQUEUE(q, h, v)
    for p ← PATIENCE downto 0 do                          ▷ Try fast-path first
        if ENQ_FAST(q, h, v, &cell_id) then
            return
        end if
    end for
    ENQ_SLOW(q, h, v, cell_id)                            ▷ Fall back to slow-path
end function

function ENQ_FAST(q, h, v, cid)
    i ← FAA(&q → T, 1)                                    ▷ Get unique cell index
    c ← FIND_CELL(&h → tail, i)                           ▷ Locate cell, allocate segment if needed
    if CAS(c.val, ⊥, v) then                              ▷ Try to deposit value
        return true
    end if
    *cid ← i                                              ▷ Return cell index for slow-path
    return false
end function

function ENQ_SLOW(q, h, v, cell_id)
    r ← &h → enq.req
    r → val ← v                                           ▷ Publish value
    r → state ← (1, cell_id)                              ▷ Set pending=1, id=cell_id
    tmp_tail ← h → tail
    repeat
        i ← FAA(&q → T, 1)
        c ← FIND_CELL(&tmp_tail, i)
        if CAS(c → enq, ⊥_e, r) and c.val = ⊥ then        ▷ Reserve cell
            TRY_TO_CLAIM_REQ(&r → state, id, i)           ▷ Claim request for this cell
            break
        end if
    until ¬r → state.pending                              ▷ Until helped by dequeuer
    id ← r → state.id                                     ▷ Get claimed cell index
    c ← FIND_CELL(&h → tail, id)
    ENQ_COMMIT(q, c, v, id)                               ▷ Write value to claimed cell
end function
```

ensures every operation completes within $O(N_{threads}^2)$ steps, as all threads will eventually help any announced operation. [32]

Verma's Queue

Uses an external helper thread that works on a dedicated core to help other processes 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 synchronisation. For consumers in algorithm 33, threads similarly create a

Algorithm 27 YMC Queue Enqueue Help Operation [37]

```
function HELP_ENQ(q, h, c, i)
    if ¬CAS(c → val, ⊥, ⊤) and c → val ≠ ⊤ then                         ▷ Value already present
        return c → val
    end if
    if c → enq = ⊥_e then                                              ▷ No request yet, find one to help
        repeat
            p ← h → enq.peer
            r ← &p → enq.req
            s ← r → state
            if h → enq.id = 0 or h → enq.id = s.id then                  ▷ Haven't helped this request
                break
            end if
            h → enq.id ← 0
            h → enq.peer ← p → next                                     ▷ Move to next peer
        until true
        if s.pending and s.id ≤ i and ¬CAS(c → enq, ⊥_e, r) then          ▷ Try to reserve cell for peer
            h → enq.id ← s.id                                           ▷ Remember we tried to help
        else
            h → enq.peer ← p → next                                     ▷ Peer doesn't need help
        end if
        if c → enq = ⊥_e then
            CAS(c → enq, ⊥_e, ⊤_e)                                        ▷ Mark no enqueue will use this cell
        end if
    end if
    if c → enq = ⊤_e then                                              ▷ No enqueue will fill this cell
        return (q → T ≤ i ? EMPTY : ⊤)                                  ▷ Check if queue was empty
    end if
    r ← c → enq                                                        ▷ Cell has enqueue request
    s ← r → state
    v ← r → val
    if s.id > i then                                                   ▷ Request unsuitable for this cell
        if c → val = ⊤ and q → T ≤ i then
            return EMPTY
        end if
    else if TRY_TO_CLAIM_REQ(&r → state, s.id, i) or (s = (0, i) and c → val = ⊤) then
        ENQ_COMMIT(q, c, v, i)                                          ▷ Help commit the value
    end if
    return c → val
end function
```

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 a round-robin fashion 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. [36]

Algorithm 28 YMC Queue Dequeue Operation [37]

```
function DEQUEUE(q, h)
    for p ← PATIENCE downto 0 do                          ▷ Try fast-path first
        v ← DEQ_FAST(q, h, &cell_id)
        if v ≠ ⊤ then break
        end if
    end for
    if v = ⊤ then                                          ▷ Fast-path failed
        v ← DEQ_SLOW(q, h, cell_id)                        ▷ Use slow-path
    end if
    if v ≠ EMPTY then                                      ▷ Got value, help peer dequeue
        HELP_DEQ(q, h, h → deq.peer)
        h → deq.peer ← h → deq.peer → next
    end if
    return v
end function

function DEQ_FAST(q, h, id)
    i ← FAA(&q → H, 1)                                     ▷ Get unique cell index
    c ← FIND_CELL(&h → head, i)
    v ← HELP_ENQ(q, h, c, i)                               ▷ Try to get/help produce value
    if v = EMPTY then return EMPTY
    end if
    if v ≠ ⊤ and CAS(c → deq, ⊥_d, ⊤_d) then               ▷ Claim the value
        return v
    end if
    *id ← i                                                ▷ Return cell index for slow-path
    return ⊤
end function

function DEQ_SLOW(q, h, cid)
    r ← &h → deq.req
    r → id ← cid                                           ▷ Set request ID
    r → state ← (1, cid)                                   ▷ Publish pending request
    HELP_DEQ(q, h, h)                                      ▷ Help complete own request
    i ← r → state.idx                                      ▷ Get index where value found
    c ← FIND_CELL(&h → head, i)
    v ← c → val
    ADVANCE_END_FOR_LINEARIZABILITY(&q → H, i + 1)         ▷ Ensure linearisability
    return (v = ⊤ ? EMPTY : v)
end function
```

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 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 $2n$ whilst only using n entries at any time, with Head and Tail counters initialised 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 sCQ's `TRY_ENQ` function of algorithm 37 up to `MAX_PATIENCE` times. If unsuccessful, the

Algorithm 29 YMC Queue Dequeue Help Operation [37]

```
function HELP_DEQ(q, h, helpee)
    r ← helpee → deq.req
    s ← r → state
    id ← r → id
    if ¬s.pending or s.idx < id then return                          ▷ Request complete or invalid
    end if
    ha ← helpee → head                                      ▷ Segment pointer for announced cells
    s ← r → state                                                    ▷ Re-read after getting head
    prior ← id; i ← id; cand ← 0
    while true do
        for hc ← ha; ¬cand and s.idx = prior; do                                ▷ Find candidate
            c ← FIND_CELL(&hc, ++i)
            v ← HELP_ENQ(q, hc, c, i)
            if v = EMPTY or (v ≠ ⊤ and c → deq = ⊥_d) then                      ▷ Found candidate
                cand ← i
            else
                s ← r → state                                                ▷ Check if announced
            end if
        end for
        if cand then
            CAS(&r → state, (1, prior), (1, cand))                      ▷ Try announce candidate
            s ← r → state
        end if
        if ¬s.pending or r → id ≠ id then return                             ▷ Request completed
        end if
        c ← FIND_CELL(&ha, s.idx)                                           ▷ Get announced cell
        if c → val = ⊤ or CAS(c → deq, ⊥_d, r) or c → deq = r then
            CAS(&r → state, s, (0, s.idx))                                    ▷ Complete request
            return
        end if
        prior ← s.idx
        if s.idx ≥ i then                                               ▷ Announced cell is ahead
            cand ← 0; i ← s.idx                                                   ▷ Jump forward
        end if
    end while
end function
```

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 synchronisation until successful. For consumers in algorithm 34, after checking for an empty queue at lines 2 to 4 and helping others at line 5, threads attempt the fast path using `TRY_DEQ` 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 synchronisation to ensure the dequeue completes. Results are gathered from the slow path in lines 25 to 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

Algorithm 30 Feldman-Dechev Queue's Enqueue Operation [32]

```
function ENQUEUE(val)
    TRYHELPANOTHER
    fails ← 0
    while true do
        if ISFULL then return false
        end if
        seqid ← NEXTTAILSEQ                                   ▷ FAA on tail
        pos ← seqid mod capacity
        n_node ← new ElemNode(seqid, val)
        while true do
            if fails++ = MAX_FAILS then
                op ← new EnqueueOp(val)
                MAKEANNOUNCEMENT(op)
                return op.RESULT
            end if
            node ← buffer[pos].LOAD
            if node.op ≠ null then                            ▷ Operation record
                node.op.ASSOCIATE(node, &buffer[pos])
                continue
            else if ISSKIPPED(node) then                      ▷ Bitmarked
                break                                         ▷ Get new seqid
            else if node.seqid < seqid then
                BACKOFF
                if node = buffer[pos].LOAD then
                    if ISEMPTYNODE(node) then
                        if buffer[pos].CAS(node, n_node) then
                            return true
                        end if
                    end if
                end if
            else if node.seqid ≤ seqid and ISEMPTYNODE(node) then
                if buffer[pos].CAS(node, n_node) then
                    return true
                end if
            else                                              ▷ node.seqid > seqid or ElemNode
                break                                         ▷ Get new seqid
            end if
        end while
    end while
end function
```

then 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 whilst 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 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 $O(N_{threads}^2)$ complexity. [35]

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 [49] 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 ≤ the 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 existed.

- Bédin et al. [50] who created a queue using a theoretical atomic primitive called memory-to-memory swap, which changes two 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 existed. Whilst this is valuable 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 results, but it is still important to mention, since it shows an interesting improvement for the time complexity of wait-free queues:

- Naderibeni and Ruppert [51] 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 JPQ. Later in chapter 7 it is seen that even though polylogarithmic, a binary tree structure is not suitable for IPC over shared memory.

Algorithm 31 Feldman-Dechev Queue's Dequeue Operation [32]

```
function Dequeue(&result)
    TryHelpAnother
    fails ← 0
    while true do
        if IsEmpty then return false
        end if
        seqid ← NextHeadSeq                                  ▷ FAA on head
        pos ← seqid mod capacity
        n_node ← new EmptyNode(seqid + capacity)
        while true do
            if fails++ = MAX_FAILS then
                op ← new DequeueOp
                MakeAnnouncement(op)
                return op.Result(result)
            end if
            node ← buffer[pos].Load
            if node.op ≠ null then
                node.op.Associate(node, &buffer[pos])
                continue
            else if IsSkipped(node) and IsEmptyNode(node) then
                if buffer[pos].CAS(node, n_node) then
                    break
                end if
            else if seqid > node.seqid then                  ▷ Delayed element
                Backoff
                if node = buffer[pos].Load then
                    if IsEmptyNode(node) then
                        if buffer[pos].CAS(node, n_node) then
                            break
                        end if
                    else
                        SetSkipped(&buffer[pos])             ▷ Bitmark
                    end if
                end if
            else if seqid < node.seqid then
                break                                        ▷ Get new seqid
            else                                             ▷ seqid = node.seqid
                if IsElemNode(node) then
                    if IsSkipped(node) then
                        n_node ← SetSkipped(n_node)
                    end if
                    if buffer[pos].CAS(node, n_node) then
                        result ← node.value
                        return true
                    end if
                else                                         ▷ EmptyNode with matching seqid
                    Backoff
                    if node = buffer[pos].Load then
                        if buffer[pos].CAS(node, n_node) then
                            break
                        end if
                    end if
                end if
            end if
        end while
    end while
end function
```

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

```
function EnqueueOp::Associate(node, address)
    success ← helper.CAS(null, node)
    if success or helper.Load = node then
        node.op.Store(null)                          ▷ Remove op reference
    else
        n_node ← new EmptyNode(node.seqid)
        if not address.CAS(node, n_node) then
            node ← SetSkipped(node)
            if address.Load = node then
                n_node ← SetSkipped(n_node)
                address.CAS(node, n_node)
            end if
        end if
    end if
end function

function DequeueOp::Associate(node, address)
    success ← helper.CAS(null, node)
    if success or helper.Load = node then
        n_node ← new EmptyNode(node.seqid + capacity)
        if not address.CAS(node, n_node) then
            node ← SetSkipped(node)
            if address.Load = node then
                n_node ← SetSkipped(n_node)
                address.CAS(node, n_node)
            end if
        end if
    else
        node.op.Store(null)
    end if
end function

function TryHelpAnother
                                 ▷ Check announcement table and help one operation
    helpIdx ← thread-local helping index
    op ← announcementTable[helpIdx]
    if op ≠ null and op.InProgress then
        if op is EnqueueOp then
            WaitFreeEnqueue(op)
        else
            WaitFreeDequeue(op)
        end if
    end if
    helpIdx ← (helpIdx + 1) mod MAX_THREADS
end function
```

Algorithm 33 Verma's Queue Operations [36]

```
function DEQUEUE
    id ← GETTHREADID                                  ▷ Get worker's unique ID
    req ← CREATEREQUEST(null, DEQUEUE)                ▷ Create dequeue request
    stateArr[id] ← req                                ▷ Place request in dedicated slot
    while ¬req.isCompleted do                         ▷ Wait for helper to process
        wait
    end while
    stateArr[id] ← null                               ▷ Clear request from state array
    return req.e                                      ▷ Return dequeued element
end function

function ENQUEUE(e)
    id ← GETTHREADID                                  ▷ Get worker's unique ID
    req ← CREATEREQUEST(e, ENQUEUE)                   ▷ Create enqueue request with element
    stateArr[id] ← req                                ▷ Place request in dedicated slot
    while ¬req.isCompleted do                         ▷ Wait for helper to process
        wait
    end while
    stateArr[id] ← null                               ▷ Clear request from state array
    return true
end function

function HELPER
    id ← 0                                            ▷ Start at first worker slot
    while true do                                     ▷ Continuous helper loop
        req ← stateArr[id]                            ▷ Check current slot for request
        if req ≠ null ∧ ¬req.isCompleted then         ▷ Found pending request
            if req.operation = ENQUEUE then
                n ← new Node(req.e)                   ▷ Create new node
                tail.next ← n                         ▷ Append to tail
                tail ← n                              ▷ Update tail reference
                size ← size + 1
                req.isCompleted ← true                ▷ Mark request complete
            else if req.operation = DEQUEUE then
                if head.next = null then              ▷ Queue is empty
                    req.e ← null
                    req.isCompleted ← true
                else
                    n ← head.next                     ▷ Get first element
                    head.next ← n.next                ▷ Remove from queue
                    if n.next = null then             ▷ Queue now empty
                        tail ← head                   ▷ Reset tail
                    end if
                    req.e ← n.e                       ▷ Store dequeued value
                    size ← size − 1
                    req.isCompleted ← true
                end if
            end if
        end if
        id ← (id + 1) mod workers                     ▷ Round-robin to next slot
    end while
end function
```

Algorithm 34 wCQ's Operations [35]

```
function DEQUEUE_WCQ
    if LOAD(&Threshold) < 0 then
        return ∅                                        ▷ Empty
    end if
    HELP_THREADS
                                                        ▷ Fast path (SCQ)
    count ← MAX_PATIENCE
    while − − count ≠ 0 do
        idx
        head ← TRY_DEQ(&idx)
        if head = OK then return idx
        end if
    end while
                                                        ▷ Slow path (wCQ)
    r ← &Record[TID]
    seq ← r.seq1
    r.localHead ← head
    r.initHead ← head
    r.enqueue ← false
    r.seq2 ← seq
    r.pending ← true
    DEQUEUE_SLOW(head, r)
    r.pending ← false
    r.seq1 ← seq + 1
                                                        ▷ Get slow-path results
    h ← COUNTER(r.localHead)
    j ← CACHE_REMAP(h mod 2n)
    Ent ← LOAD(&Entry[j].Value)
    if Ent.Cycle = CYCLE(h) and Ent.Index ≠ ⊥ then
        CONSUME(h, j, Ent)
        return Ent.Index
    end if
    return ∅
end function

function ENQUEUE_WCQ(index)
    HELP_THREADS
                                                        ▷ Fast path (SCQ)
    count ← MAX_PATIENCE
    while − − count ≠ 0 do
        tail ← TRY_ENQ(index)
        if tail = OK then return true
        end if
    end while
                                                        ▷ Slow path (wCQ)
    r ← &Record[TID]
    seq ← r.seq1
    r.localTail ← tail
    r.initTail ← tail
    r.index ← index
    r.enqueue ← true
    r.seq2 ← seq
    r.pending ← true
    ENQUEUE_SLOW(tail, index, r)
    r.pending ← false
    r.seq1 ← seq + 1
end function
```

Algorithm 35 wCQ's Helper Functions [35]

```
function HELP_THREADS
    r ← &Record[TID]
    if − − r.nextCheck ≠ 0 then return
    end if
    thr ← &Record[r.nextTid]
    if thr.pending then
        if thr.enqueue then
            HELP_ENQUEUE(thr)
        else
            HELP_DEQUEUE(thr)
        end if
    end if
    r.nextCheck ← HELP_DELAY
    r.nextTid ← (r.nextTid + 1) mod NUM_THRDS
end function

function SLOW_F&A(globalp, local, v, thld)
    phase2 ← &Record[TID].phase2
    repeat
        cnt ← LOAD_GLOBAL_HELP_PHASE2(globalp, local)
        if cnt = ∅ or !CAS(local, ∗v, cnt|INC) then
            ∗v ← ∗local
            if ∗v&FIN then return false
            end if
            if !(∗v&INC) then return true
            end if
            cnt ← COUNTER(∗v)
        else
            ∗v ← cnt|INC
        end if
        PREPARE_PHASE2(phase2, local, cnt)
    until CAS2(globalp, {cnt, null}, {cnt + 1, phase2})
    if thld then F&A(thld, −1)
    end if
    CAS(local, cnt|INC, cnt)
    CAS2(globalp, {cnt + 1, phase2}, {cnt + 1, null})
    ∗v ← cnt
    return true
end function
```

Algorithm 36 wCQ's Data Structures [35]

```
struct phase2rec_t {
    seq1 : int = 1
    *local : int
    cnt : int
    seq2 : int = 0
}

struct entpair_t {
    Note : int = -1
    Value : entry_t = { .Cycle=0, .IsSafe=1, .Enq=1, .Index=⊥ }
}

Entry[2n] : entpair_t

struct thrdrec_t {
                                                    ▷ === Private Fields ===
    nextCheck : int = HELP_DELAY
    nextTid : int                                   ▷ Thread ID
                                                    ▷ === Shared Fields ===
    phase2 : phase2rec_t                            ▷ Phase 2
    seq1 : int = 1
    enqueue : bool
    pending : bool = false
    localTail, initTail : int
    localHead, initHead : int
    index : int
    seq2 : int = 0
}

Record[NUM_THRDS] : thrdrec_t
Threshold : int = -1                                ▷ Empty wCQ
Tail : int = 2n, Head : int = 2n
```

Algorithm 37 Lock-free Circular Queue (SCQ): Enqueue Operations [35]

```
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 [35]

```
function TRY_DEQ(*index)
    H ← F&A(&Head, 1)
    j ← CACHE_REMAP(H mod 2n)
    E ← LOAD(&Entry[j])
    if E.Cycle = CYCLE(H) then
        CONSUME(H, j, E)
        *index ← E.Index
        return OK                                  ▷ Success
    end if
    New ← {E.Cycle, 0, E.Index}
    if E.Index = ⊥ or ⊥_c then
        New ← {CYCLE(H), E.IsSafe, ⊥}
    end if
    if E.Cycle < CYCLE(H) then
        if !CAS(&Entry[j], E, New) then
            goto 33
        end if
        T ← LOAD(&Tail)                            ▷ Exit if
        if T ≤ H + 1 then                          ▷ empty
            CATCHUP(T, H + 1)
        end if
        F&A(&Threshold, −1)
        *index ← ∅                                 ▷ Empty
        return OK                                  ▷ Success
    end if
    if F&A(&Threshold, −1) ≤ 0 then
        *index ← ∅                                 ▷ Empty
        return OK                                  ▷ Success
    end if
    return H                                       ▷ Try again
end function

function DEQUEUE_SCQ
    if LOAD(&Threshold) < 0 then
        return ∅                                   ▷ Empty
    end if
    while TRY_DEQ(&idx) ≠ OK do
                                                   ▷ Try again
    end while
    return idx
end function

function CONSUME(h, j, e)
    OR(&Entry[j], {0, 0, ⊥_c})
end function
```

Algorithm 39 CAS2 implementation for wCQ using LL/SC [35]

```
function CAS2_Value(Var: entpair_t*, Expect: entpair_t, New: entpair_t)
    Prev.Value ← LL(&Var → Value)
    Prev.Note ← Load(&Var → Note)
    if Prev ≠ Expect then
        return false
    end if
    return SC(&Var → Value, New.Value)
end function

function CAS2_Note(Var: entpair_t*, Expect: entpair_t, New: entpair_t)
    Prev.Note ← LL(&Var → Note)
    Prev.Value ← Load(&Var → Value)
    if Prev ≠ Expect then
        return false
    end if
    return SC(&Var → Note, New.Note)
end function
```

6. Implementation

After the wait-free algorithms were identified and analysed, the next sub-goal to find the best wait-free algorithms is to implement them like already said in section 1.2. The algorithms were implemented and published in the accompanying GitHub repository in [47], where complete implementation of the algorithms can be found. This chapter will present the implementation details of the concurrent queue algorithms seen in chapter 5 in Rust, focusing on the adaptations necessary for IPC over shared memory. While the algorithmic logic of each queue has already been discussed in chapter 5, the implementation required slight deviations to support IPC over shared memory. This includes ensuring correct cache alignment, correctly implementing various atomic primitives, and correctly implementing the logic for shared memory support using short example snippets of the actual Rust implementation from the GitHub repository or some general examples. This does not include showing again the same algorithmic logic, just in an actual programming language instead of pseudocode, since that would be redundant. This chapter will give the necessary details to understand how to implement the algorithms in Rust and how to adapt them for IPC over shared memory.

6.1. Shared Memory Management for Inter-Process Communication (IPC)

IPC through shared memory requires slightly different approaches compared to multi-threaded heap-based communication. 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 buffer allocation from a pre-allocated memory pool for all dynamic data structures. Threads, on the other hand, can use the process-private heap for dynamic memory allocation, whose memory layout is shared between threads on the same process, but not between processes.

6.1.1. Shared Memory Size Calculation

Each queue provides a method to calculate the exact shared memory size required. This calculation determines how much memory to allocate from the operating system. The following examples from BQueue and wCQ demonstrate both simple and complex size calculations as needed:

Listing 6.1: Shared memory size calculation methods

```
1  // From BQueue - simple calculation
2  pub const fn shared_size(capacity: usize) -> usize {
3      mem::size_of::<Self>()                          // Queue structure
4          + capacity * mem::size_of::<MaybeUninit<T>>()   // Data storage
5          + capacity * mem::size_of::<AtomicBool>()       // Validity flags
6  }
7
8  // From WCQueue - complex layout calculation
9  fn layout(num_threads: usize, num_indices: usize) -> (Layout, [usize; 4]) {
10     let ring_size = num_indices.next_power_of_two();
11     let capacity = ring_size * 2;
12
13     let root = Layout::new::<Self>();
14     let (l_aq_entries, o_aq_entries) = root
15         .extend(Layout::array::<EntryPair>(capacity).unwrap())
16         .unwrap();
17     let (l_fq_entries, o_fq_entries) = l_aq_entries
18         .extend(Layout::array::<EntryPair>(capacity).unwrap())
19         .unwrap();
20     let (l_records, o_records) = l_fq_entries
21         .extend(Layout::array::<ThreadRecord>(num_threads).unwrap())
22         .unwrap();
23     let (l_final, o_data) = l_records
24         .extend(Layout::array::<DataEntry<T>>(num_indices).unwrap())
25         .unwrap();
26
27     (l_final.pad_to_align(), [o_aq_entries, o_fq_entries, o_records, o_data])
28 }
29
30 pub fn shared_size(num_threads: usize) -> usize {
31     let (_layout, _offsets) = Self::layout(num_threads, num_indices);
32     _layout.size()  // Total bytes needed for mmap
33 }
```

The size calculation accounts for all components, including alignment padding. Some queues like wCQ use Rust's `Layout` API shown in lines 9 to 26 to ensure proper alignment and correctly calculated offsets. For simple queues like BQueue, manual calculation like in lines 2 to 6 suffices.

6.1.2. Shared Memory Allocation

Once the required size is calculated, the following functions in listing 6.2 used in all benchmarks demonstrate how to allocate and deallocate the required shared memory regions using `mmap`:

Listing 6.2: Shared memory allocation using mmap

```
unsafe fn map_shared(bytes: usize) -> *mut u8 {
    let ptr = libc::mmap(
```

```
3           ptr::null_mut(),
4           bytes,                                      // Size from shared_size()
5           libc::PROT_READ | libc::PROT_WRITE,
6           libc::MAP_SHARED | libc::MAP_ANONYMOUS,    // Shared between processes
7           -1,
8           0,
9       );
10      if ptr == libc::MAP_FAILED {
11          panic!("mmap failed: {}", std::io::Error::last_os_error());
12      }
13      ptr.cast()
14  }
15
16  // Cleanup function
17  unsafe fn unmap_shared(ptr: *mut u8, len: usize) {
18      if libc::munmap(ptr.cast(), len) == -1 {
19          panic!("munmap failed: {}", std::io::Error::last_os_error());
20      }
21  }
```

The `MAP_SHARED` flag ensures that modifications to the memory region are visible to all processes that map it. The `MAP_ANONYMOUS` flag creates memory not backed by a file. The `bytes` parameter on line 4 comes directly from the `shared_size()` calculation earlier.

6.1.3. Memory Layout and Initialisation

After allocating the shared memory region, the components needed by the queue implementations need to be initialised. All queue implementations follow a consistent pattern for shared memory initialisation. The initialisation function receives the pre-allocated memory pointer and organises components within that memory region. The most complex example, the WCQueue from section 5.2.4, demonstrates how multiple components are laid out in shared memory with proper alignment:

Listing 6.3: Memory layout initialisation in WCQueue

```
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);
```

```
11      let aq_entries = mem.add(current_offset) as *mut EntryPair;
12      current_offset += capacity * mem::size_of::<EntryPair>();
13
14      // Initialise structures in-place
15      ptr::write(q_ptr, Self {
16          aq_entries_offset: current_offset,
17          base_ptr: mem,  // Store base pointer for offset calculations
18          // Store offsets instead of pointers
19      });
20
21      &mut *q_ptr
22  }
```

The alignment calculation in line 5 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 in line 16, ensuring position independence. When accessing these components later, the offset is added to the base pointer, as demonstrated in listing 6.4.

Listing 6.4: Position-independent component access

```
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)
}
```

6.1.4. Node Allocation from Pre-allocated Memory Pools

Some queues as seen in chapter 5 use dynamic memory allocation within the process-private heap. For the IPC over shared memory use case of this thesis that is not possible, because in shared memory dynamic memory allocations or deallocations with `malloc` or `free` are not possible, since that allocates from process-private heaps that are not available for other processes. Therefore, all queues that would normally allocate memory dynamically have been adapted to allocate from the initialised pre-allocated memory pool. As an example, the Jiffy Queue algorithm from section 5.2.2 shows a dynamic heap allocation method to allocate new buffers to insert them into the linked list. This was adapted to IPC over shared memory in Rust by a pool allocation system with free lists that is similarly implemented for all other queues needing dynamic memory allocation, as shown in listing 6.5.

Listing 6.5: Lock-free memory pool allocation

```
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;
```

```rust
7         }
8 
9         let next_free = (*(free_head as *mut AtomicPtr<Node<T>>))
10             .load(Ordering::Acquire);
11 
12         if self.node_array_slice_free_list_head.compare_exchange(
13             free_head,
14             next_free,
15             Ordering::AcqRel,
16             Ordering::Acquire
17         ).is_ok() {
18             return free_head;
19         }
20     }
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 }
```

This implementation maintains a free list using CAS operations in lines 12 to 16. When the free list is empty, it falls back to allocate from the pre-allocated pool seen in lines 23 and 24. The `fetch_add` operation atomically increments the allocation index, ensuring process-safe allocation. If the pool is exhausted, the operation fails by returning a null pointer in line 30.

6.2. Cache Line Optimisation

Processors transfer data between cores in cache line units, explained in section 5.2.1. When multiple processes access data on the same cache line, even if different variables, the cache coherence protocol causes the cache line to bounce between cores, degrading execution times.

6.2.1. Explicit Cache Line Padding

In chapter 5, multiple queues were shown that describe separating the cache line. To show how this is done in rust the BLQ explained in section 5.2.1 will be taken to show how to explicitly separate the cache lines, as shown in listing 6.6.

Listing 6.6: Cache line separation in BlqQueue

```rust
const CACHE_LINE_SIZE: usize = 64;
```

```rust
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 {
12     read_shadow: usize,      // Local copy to avoid false sharing
13     write_priv: usize,       // Producer-only write position
14 }
15
16 #[repr(C, align(64))]
17 struct ConsumerPrivate {
18     write_shadow: usize,     // Local copy to avoid false sharing
19     read_priv: usize,        // Consumer-only read position
20 }
```

The `#[repr(C)]` attribute ensures C-compatible memory layout, while `#[repr(align(64))]` forces the structure to start at a cache line boundary, shown in lines 4 and 10. Although `SharedIndices` contains only two `usize` values with 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` in line 12 and `write_shadow` in line 18 ensure that producer and consumer work on different cache lines, preventing the cache lines from bouncing between the producer process and consumer process.

Similarly, all queues that need this kind of cache line separation use this pattern to ensure that the producer and consumer do not share cache lines, preventing false sharing, leading to cache lines bouncing between cores.

6.2.2. Manual Padding Arrays

For structures where alignment alone is insufficient, manual padding arrays provide a solution, as demonstrated in listing 6.7. This is used in all queues needing manual padding. As an example, the implementation of the David queue explained in section 5.2.3 uses manual padding.

Listing 6.7: Manual padding for exact cache line control

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

#[repr(C, align(64))]
```

```
8 struct Node<T> {
9     val: Option<T>,
10     next: AtomicPtr<Node<T>>,
11     _padding: [u8; CACHE_LINE_SIZE - 24], // Fill remaining cache line
12 }
```

The padding array size is calculated to fill the remainder of the cache line seen in lines 4 and 11. 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, as required by the DavidQueue implementation from section 5.2.3.

6.3. Atomic Primitives Implementation

The algorithms in chapter 5 all use different kinds of atomic primitives. To implement them, rust provides a set of atomic operations with explicit memory ordering semantics, allowing control over synchronisation order. This section shows how in general the atomic operations from chapter 5 are implemented in Rust across all queues, explained with specific examples. In rust, atomic primitives can only be called on atomic types. Hence variables that are used in atomic operations must be defined as atomic types.

6.3.1. Fetch and Add (FAA)

The rust implementations of DQueue, BQueue and wCQ explained in chapter 5 are a good example to understand how to implement FAA called `fetch_add(v, ordering)` in rust.

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

```
1 // From Jiffy - Allocate multiple nodes at once
2 let start_node_idx = self
3     .node_arrays_next_free_node_idx
4     .fetch_add(nodes_needed, Ordering::AcqRel);
5
6 // From BQueue - private counter with relaxed ordering
7 let idx = self.next_item.fetch_add(1, Ordering::Relaxed);
8
9 // From WCQueue - with sequential consistency for wait-free algorithm
10 let seqid = self.tail.fetch_add(1, Ordering::SeqCst);
```

In Rust, FAA is a method call on atomic types as seen in line 4. In listing 6.8, the memory ordering parameter added as the second argument after the value to add determines the synchronisation order of FAA. `Ordering::Relaxed` in line 7 provides no synchronisation, suitable for private counters. `Ordering::AcqRel` in line 4 ensures acquire semantics for the read and release semantics for the write, establishing happens-before relationships as required by the DQueue algorithm. `Ordering::SeqCst` in line 10 provides the strongest guarantees, ensuring a total order across all sequentially consistent operations, necessary for the complex wCQ. One simple solution would be to always use `Ordering::SeqCst` for all operations, but that would reduce the execution times of the algorithms in a significant way. Consequently, it is important to analyse the algorithms from chapter 5 to understand which memory ordering is needed for which operation.

6.3.2. Compare and Swap (CAS)

The implementation of wCQ shows how to implement CAS, called `compare_exchange(old_value, new_value, ordering_on_success, ordering_on_failure)` in rust, as shown in listing 6.9.

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

```
1  // Strong CAS with sequential consistency (from wcqueue)
2  match entry.value.compare_exchange(
3      packed,
4      new_packed,
5      Ordering::SeqCst,     // Success ordering
6      Ordering::SeqCst,     // Failure ordering
7  ) {
8      Ok(_) => {
9          fence(Ordering::SeqCst);  // Additional synchronisation
10         Ok(())
11     }
12     Err(current) => {
13         // Retry with current value
14     }
15 }
16
17 // Weak CAS for performance (general example)
18 match entry.value.compare_exchange_weak(
19     old_value,
20     new_value,
21     Ordering::AcqRel,
22     Ordering::Acquire,
23 ) {
24     Ok(_) => {
25         // do something on success
26     }
27     Err(current) => {
28         // do something on failure
29     }
30 }
```

The weak variant `compare_exchange_weak` beginning at line 18 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 in lines 5 and 6 and 21 and 22 specify synchronisation order for success and failure cases respectively at lines 8 and 12. This directly implements the CAS operations described in multiple algorithms.

6.3.3. Swap Operations

Swap unconditionally replaces a value and returns the previous value, implemented as `swap(new_value, ordering)`, as shown in listing 6.10. As an example, the David Queue and Drescher Queue is used.

Listing 6.10: Unconditional atomic swap operations

```
1  // From David Queue - unconditional slot update
2  unsafe fn swap(&self, new_val: usize) -> usize {
3      self.value.swap(new_val, Ordering::AcqRel)
4  }
5
6  // From Drescher Queue - atomic pointer swap
7  let prev_tail = self.tail.swap(new_node_ptr, Ordering::AcqRel);
8
9  // From DrescherQueue - simpler FAS primitive
10 let prev_tail_ptr = self.tail.swap(new_node_ptr, Ordering::AcqRel);
11 (*prev_tail_ptr).next.store(new_node_ptr, Ordering::Release);
```

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 (line 10), atomically updating the tail pointer and then linking the previous tail to the new node, exactly as specified in the Drescher algorithm in section 5.2.2.

6.3.4. Load and Store with Memory Ordering

Simple loads and stores also require consideration of memory ordering to ensure correct synchronisation according to the algorithms in chapter 5, as demonstrated in the general example of listing 6.11.

Listing 6.11: Memory ordering for loads and stores

```
1  // Acquire ordering for reading shared state
2  let tail = self.tail.load(Ordering::Acquire);
3  if tail > head {
4      // Safe to proceed - acquire ensures we see all writes before tail update
5  }
6
7  // Release ordering for publishing updates
8  unsafe { (*slot.data.get()).write(item); }  // Write data first
9  self.head.store(new_head, Ordering::Release);  // Then publish
10
11 // Sequential consistency for strong synchronisation
12 let val = entry.value.load(Ordering::SeqCst);
13 entry.value.store(new_val, Ordering::SeqCst);
```

The acquire-release pattern is particularly important. A release store in line 9 synchronises with an acquire load in line 2 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 used in all queues to ensure correct ordering of the producer-consumer relationship, implementing the memory barriers described in algorithms like Lamport Queue (section 5.2.1) and others.

6.3.5. Memory Fences

To still ensure correct data ordering without any memory operation, rust provides memory fences seen in the wCQ rust implementation, as shown in listing 6.12.

Listing 6.12: Explicit memory fence usage

```
1  // From WCQueue - ensuring visibility across operations
2  fence(Ordering::SeqCst);
3  let packed = entry.value.load(Ordering::SeqCst);
4  let e = EntryPair::unpack_entry(packed);
5
6  if condition {
7      fence(Ordering::SeqCst);  // Ensure all prior operations complete
8
9      match entry.value.compare_exchange_weak(
10         packed,
11         new_packed,
12         Ordering::SeqCst,
13         Ordering::SeqCst,
14     ) {
15         Ok(_) => {
16             fence(Ordering::SeqCst);  // Ensure visibility before proceeding
17         }
18     }
19 }
```

Fences ensure ordering between operations that might not otherwise synchronise. The wCQ implementation uses sequential consistency fences (lines 2, 7, 16) to ensure correctness in its complex algorithm. These kinds of fences are used in all queue implementations since for IPC this was necessary for correctness.

6.3.6. Versioned Compare and Swap (CAS) (Simulating Load-Linked and Store-Conditional (LL/SC))

Some algorithms assume LL/SC primitives, which x86-64 does not provide. The JPQ from section 5.2.2 simulates LL/SC using versioned CAS, as shown in listing 6.13.

Listing 6.13: Versioned CAS for LL/SC simulation

```
#[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)
```

```
7  }
8
9  impl CompactMinInfo {
10     fn to_u64(self) -> u64 {
11         ((self.version as u64) << 32)
12             | ((self.ts_val as u64) << 16)
13             | ((self.ts_pid as u64) << 8)
14             | (self.leaf_idx as u64)
15     }
16 }
17
18 unsafe fn cas_min_info(&self, old_compact: CompactMinInfo,
19                        new_min_info: MinInfo) -> bool {
20     // Increment version on every update
21     let new_compact = CompactMinInfo::from_min_info(
22         new_min_info,
23         old_compact.version + 1
24     );
25
26     self.compact_min_info.compare_exchange(
27         old_compact.to_u64(),
28         new_compact.to_u64(),
29         Ordering::AcqRel,
30         Ordering::Acquire
31     ).is_ok()
32 }
```

The version counter in line 3 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 seen in line 23, even if the logical value returns to a previous state, the version ensures the CAS will fail, simulating LL/SC semantics as required by the JPQ algorithm. This is done in every queue rust implementation that requires LL/SC semantics.

6.3.7. Unsafe Blocks

Rust's memory safety guarantees prevent data races by ensuring that either multiple readers or a single writer can access data at any time. However, the wait-free algorithms in chapter 5 require bypassing these restrictions. Through the use of `unsafe` blocks, the programmer indicates to the Rust compiler that the block's memory safety is handled by the implementation itself. The `try_enq_inner` function of wCQ's Rust implementation in listing 6.14 demonstrates why `unsafe` blocks are necessary.

Listing 6.14: Wait-free synchronisation requiring unsafe

```
    // Multiple threads may execute this concurrently
    unsafe fn try_enq_inner(&self, wq: &InnerWCQ, entries_offset: usize,
                            index: usize) -> Result<(), u64> {
        let tail = wq.tail.cnt.fetch_add(1, Ordering::AcqRel);
        let j = Self::cache_remap(tail as usize, wq.capacity);

```

```
7           let entry = self.get_entry(wq, entries_offset, j);
8           loop {
9               let packed = entry.value.load(Ordering::Acquire);
10              let e = EntryPair::unpack_entry(packed);
11
12              // Check if slot is available
13              if e.cycle < Self::cycle(tail, wq.ring_size) &&
14                  (e.is_safe || wq.head.cnt.load(Ordering::Acquire) <= tail)
    ↪ &&
15                  (e.index == IDX_EMPTY || e.index == IDX_BOTTOM) {
16
17                  // Attempt to claim slot with CAS
18                  match entry.value.compare_exchange_weak(
19                      packed,
20                      new_packed,
21                      Ordering::SeqCst,
22                      Ordering::SeqCst,
23                  ) {
24                      Ok(_) => return Ok(()),
25                      Err(_) => continue,   // Retry
26                  }
27              }
28          }
29      }
30
31      // get_entry uses raw pointer arithmetic
32      unsafe fn get_entry(&self, _wq: &InnerWCQ, entries_offset: usize,
33                          idx: usize) -> &EntryPair {
34          let entries = self.base_ptr.add(entries_offset) as *const
    ↪ EntryPair;
35          &*entries.add(idx)   // Dereference raw pointer
36      }
```

One reason for an `unsafe` block is raw pointer dereferencing seen in lines 32 to 34. The `get_entry` function performs pointer arithmetic on `base_ptr` and dereferences the result, which is needed for shared memory. The compiler cannot verify that the calculated address is valid or that no data races occur. The Rust compiler also does not allow concurrent access of multiple writer threads or processes. `try_enq_inner` beginning at line 2 is implemented so that multiple processes can simultaneously call it. Each process atomically increments the tail to get a unique position in line 4, then accesses potentially the same entry due to cache remapping in lines 5 and 7, and finally attempts to modify the entry in line 18. The Rust compiler cannot verify that this access is safe, so the developer of this implementation has to indicate to the compiler that this is safe by using an `unsafe` block.

6.4. Validation

Additionally, ensuring the correctness of the implemented algorithms is also part of the objective to ensure the correctness of the implemented algorithms; unit and miri tests were performed. The unit tests validate the basic functionality of each queue, ensuring that operations like enqueue and dequeue work as expected. Miri tests were used to check for undefined behaviour in concurrent scenarios, ensuring that the algorithms behave correctly under extreme contention. This section will generally describe how the tests were implemented and what they validate, without going into the details of each test case.

6.4.1. Basic Operations

Every queue implementation was validated for fundamental operations including initialisation, push, pop, and state queries (`is_empty`, `is_full`). For example, the basic operation test pattern was implemented as shown in listing 6.15:

Listing 6.15: Basic operation test pattern

```
1  // Test empty queue state
2  assert!(queue.is_empty());
3  assert!(queue.pop().is_err());
4
5  // Test single element operations
6  queue.push(42).unwrap();
7  assert!(!queue.is_empty());
8  assert_eq!(queue.pop().unwrap(), 42);
9  assert!(queue.is_empty());
10
11 // Test multiple elements
12 for i in 0..10 {
13     queue.push(i).unwrap();
14 }
15 for i in 0..10 {
16     assert_eq!(queue.pop().unwrap(), i);
17 }
```

Lines 2 to 3 in listing 6.15 test the initial empty state and check that pop operations fail correctly on empty queues. The single-element test in lines 6 to 9 tests the state change from empty to non-empty and back, with line 8 testing that the pushed value is correctly retrieved. Lines 12 to 17 test that FIFO ordering is maintained for multiple elements.

6.4.2. Capacity and Boundary Tests

Capacity limits also had to be tested, so it can be ensured that the capacities work correctly. Special attention was given to queues with buffering mechanisms like BIFFQ and MultiPush queue, which required explicit flushing operations.

Listing 6.16: Capacity limit test pattern

```
// Test pushing up to capacity
let mut pushed = 0;
for i in 0..capacity {
    match queue.push(i) {
```

```
5             Ok(_) => pushed += 1,
6             Err(_) => break,
7         }
8     }
9     assert!(pushed > 0, "Should push at least one item");
10
11    // Verify queue rejects items when full
12    assert!(!queue.available() || queue.push(999).is_err());
13
14    // Test space recovery after pop
15    if pushed > 0 {
16        assert!(queue.pop().is_ok());
17        assert!(queue.available());
18        assert!(queue.push(888).is_ok());
19    }
```

In listing 6.16, lines 3 to 8 test filling the queue to capacity, counting successful pushes. Line 9 tests that at least one item could be pushed. Line 12 tests that a full queue either reports no available space or rejects push attempts. Lines 15 to 19 test that after removing an element, space becomes available for new items.

For queues with buffering mechanisms, additional testing was required:

Listing 6.17: Buffered queue capacity test

```
// BiffQ attempts to push beyond capacity
for i in 0..BIFFQ_CAPACITY + 100 {
    match queue.push(i) {
        Ok(_) => pushed_total += 1,
        Err(_) => {
            // Flush local buffer to main queue
            let _ = queue.flush_producer_buffer();
            if queue.push(i).is_err() {
                break;  // No space in main queue
            } else {
                pushed_total += 1;
            }
        }
    }
    // Periodic flushing every 32 items
    if i % 32 == 31 {
        let _ = queue.flush_producer_buffer();
    }
}
// Final flush to ensure all buffered items are visible
let _ = queue.flush_producer_buffer();

// Verify capacity behaviour based on how full the queue got
if pushed_total >= BIFFQ_CAPACITY - 32 {
    // Nearly full: test pop/push with limited space
    assert!(queue.pop().is_ok());
    // Try to push after making space
```

```
28         for _ in 0..10 {
29             let _ = queue.flush_producer_buffer();
30             if queue.push(99999).is_ok() {
31                 break;
32             }
33             let _ = queue.pop(); // Make more space
34         }
35     } else {
36         // Partially full: verify basic push/pop works
37         assert!(queue.pop().is_ok());
38         assert!(queue.push(99999).is_ok());
39     }
```

The BIFFQ test in listing 6.17 tests the buffering system. When push fails in line 5, line 7 flushes the local buffer to the main queue. If the retry in line 8 still fails, it means the main queue lacks space. Lines 16 to 18 periodically flush every 32 items (the buffer size). The test in lines 24 to 39 checks different behaviours based on how full the queue is: lines 24 to 33 test the case where the main queue might not have room for a complete buffer flush, while lines 35 to 39 test basic operations when the queue is only partially full.

6.4.3. Memory Alignment Verification

Since the implementations target shared memory with specific alignment requirements, tests as in listing 6.18 verified proper memory alignment for all queue structures:

Listing 6.18: Memory alignment verification test

```
// Allocate memory with specific alignment
let layout = Layout::from_size_align(size, alignment)
    .expect("Invalid␣layout");
let ptr = unsafe { alloc_zeroed(layout) };

// Verify pointer alignment
assert_eq!(
    ptr as usize % alignment, 0,
    "Memory␣not␣aligned␣to␣{}␣bytes", alignment
);

// Initialise queue with aligned memory
let queue = unsafe {
    <$queue_type>::init_in_shared(ptr, capacity)
};

// For YangCrummeyQueue requiring 128-byte alignment
let queue_offset = sync_size;
let queue_offset_aligned = (queue_offset + 127) & !127;
let queue_ptr = unsafe { shm_ptr.add(queue_offset_aligned) };
assert_eq!(
    queue_ptr as usize % 128, 0,
    "Queue␣not␣properly␣aligned␣to␣128␣bytes"
```

```
24 );
```

Lines 2 to 4 in listing 6.18 create memory with the required alignment and allocate it so that lines 7 to 10 can test that the allocation has the correct alignment using modulo arithmetic. For YMC, lines 18 and 19 calculate the aligned offset, where `(offset + 127) & !127` rounds up to the next 128-byte boundary. Lines 21 to 24 test that the final queue pointer has the required 128-byte alignment.

Position-independent addressing was verified in listing 6.19 through shared memory tests:

Listing 6.19: Position-independent addressing test

```
1  // Map shared memory at arbitrary address
2  let shm_ptr = unsafe {
3      libc::mmap(
4          std::ptr::null_mut(),  // Let OS choose address
5          size,
6          libc::PROT_READ | libc::PROT_WRITE,
7          libc::MAP_SHARED | libc::MAP_ANONYMOUS,
8          -1,
9          0,
10     ) as *mut u8
11 };
12
13 // Initialise queue - must work at any address
14 let queue = unsafe {
15     Queue::init_in_shared(shm_ptr, capacity)
16 };
17
18 // Verify queue operates correctly regardless of base address
19 queue.push(42).unwrap();
20 assert_eq!(queue.pop().unwrap(), 42);
```

Line 4 in listing 6.19 passes null to let the OS choose the memory address and lines 6 and 7 specify shared anonymous memory for inter-process access. Lines 14 and 15 initialise the queue at any address. Lines 19 and 20 test that the queue works correctly regardless of its memory location.

Memory Pool Management

The pre-allocated memory pools of the queues were tested for correct allocation, recycling, and pool exhaustion handling:

Listing 6.20: Memory pool management test

```
// Test pool allocation and recycling
unsafe {
    // Allocate from pool
    let seg1: *mut Segment<i32> = queue.new_segment(1);
    assert!(!seg1.is_null());
    assert_eq!((*seg1).id, 1);

```

```
8         // Return to pool
9         queue.release_segment_to_pool(seg1);
10
11        // Test pool exhaustion
12        let mut segments = vec![];
13        for i in 2..segment_pool_capacity as u64 {
14            let seg = queue.new_segment(i);
15            if !seg.is_null() {
16                segments.push(seg);
17            } else {
18                break;  // Pool exhausted
19            }
20        }
21
22        // Return all to pool
23        for seg in segments {
24            queue.release_segment_to_pool(seg);
25        }
26    }
27
28    // Test free list reuse in Jiffy Queue
29    let free_head = self.node_array_slice_free_list_head
30        .load(Ordering::Acquire);
31    if !free_head.is_null() {
32        // Reuse from free list
33        let next_free = (*(free_head as *mut AtomicPtr<Node<T>>))
34            .load(Ordering::Acquire);
35
36        if self.node_array_slice_free_list_head.compare_exchange(
37            free_head,
38            next_free,
39            Ordering::AcqRel,
40            Ordering::Acquire
41        ).is_ok() {
42            return free_head;  // Successfully reused
43        }
44    }
```

Lines 4 to 6 in listing 6.20 test basic allocation from the pool and test that the segment is properly initialised. Line 9 tests returning the segment to the pool. Lines 12 to 20 test pool exhaustion by allocating until `new_segment` returns null in line 18. Lines 23 to 25 test that segments can be returned to the pool. Lines 29 to 44 test the free list implementation in Jiffy Queue, where lines 36 to 41 test atomic removal of the free list head using `compare_exchange`. Since we only test that the allocations work and can be used, assertions were not used, since only correct behaviour is tested.

6.4.4. Concurrent Operation Tests

Concurrent tests validated the correctness of queue operations under contention. Different kinds of tests were implemented to stress the queues in various ways:

Stress Tests

In listing 6.21, high-contention scenarios were tested with multiple producers and consumers operating simultaneously:

Listing 6.21: High-contention stress test

```
// Stress test with multiple concurrent producers and consumers
let num_threads = 4;
let items_per_thread = 1000;
let produced = Arc::new(AtomicUsize::new(0));
let consumed = Arc::new(AtomicUsize::new(0));
let done = Arc::new(AtomicBool::new(false));

// Spawn producer threads
for tid in 0..num_threads / 2 {
    let p = Arc::clone(&produced);
    let handle = thread::spawn(move || {
        for i in 0..items_per_thread {
            let value = tid * items_per_thread + i;
            let mut retries = 0;
            while queue.push(value, tid).is_err() && retries < 1000 {
                retries += 1;
                thread::yield_now();
            }
            if retries < 1000 {
                p.fetch_add(1, Ordering::Relaxed);
            }
        }
    });
    handles.push(handle);
}

// Spawn consumer threads
for tid in num_threads / 2..num_threads {
    let c = Arc::clone(&consumed);
    let d = Arc::clone(&done);
    let handle = thread::spawn(move || {
        let mut consecutive_failures = 0;
        loop {
            if queue.pop(tid).is_ok() {
                c.fetch_add(1, Ordering::Relaxed);
                consecutive_failures = 0;
            } else {
                consecutive_failures += 1;
                if d.load(Ordering::Relaxed) &&
```

```
40                          consecutive_failures > 100 {
41                          break;
42                      }
43                      thread::yield_now();
44                  }
45              }
46          });
47          handles.push(handle);
48  }
49
50  // Let threads run under contention
51  thread::sleep(Duration::from_millis(100));
52  done.store(true, Ordering::Relaxed);
53
54  // Verify no items lost
55  let produced_count = produced.load(Ordering::Relaxed);
56  let consumed_count = consumed.load(Ordering::Relaxed);
57  assert!(
58      consumed_count >= produced_count * 9 / 10,
59      "Should␣consume␣at␣least␣90%␣of␣produced␣items"
60  );
```

Lines 4 to 6 in listing 6.21 create atomic counters to track production and consumption across threads. Lines 15 to 17 test retry behaviour with yield to handle temporary failures. Lines 19 and 20 only count successfully pushed items. Lines 33 to 45 test the consumer loop that continues until signalled to stop and experiences many failures in lines 39 to 41. Lines 51 and 52 signal consumers to terminate. Lines 55 to 60 test that at least 90% of the produced items were consumed.

In the following listing 6.22, correct FIFO ordering was tested:

Listing 6.22: FIFO ordering verification under stress

```
// Collect all dequeued items
let mut items = Vec::new();
while let Some(item) = queue.pop() {
    items.push(item);
}

// Verify correct count
assert_eq!(items.len(), NUM_PRODUCERS * ITEMS_PER_PRODUCER);

// Sort and verify no duplicates or missing items
items.sort();
for (i, &item) in items.iter().enumerate() {
    assert_eq!(item, i, "Missing␣or␣duplicate␣items␣detected");
}

// For strict FIFO queues, verify ordering per producer
for producer_id in 0..num_producers {
    let producer_items: Vec<_> = items.iter()
        .filter(|&&x| x / 1000 == producer_id)
```

```
20         .collect();
21
22     // Items from same producer must maintain order
23     for window in producer_items.windows(2) {
24         assert!(window[0] < window[1],
25                 "FIFO␣order␣violated␣for␣producer␣{}", producer_id);
26     }
27 }
```

Lines 2 to 5 in listing 6.22 collect all items from the queue. Lines 11 to 14 test for missing or duplicate items by checking sequential values after sorting. Lines 17 to 20 filter items by producer. Lines 23 to 26 test that items from the same producer maintain their order.

6.4.5. Inter-Process Communication (IPC) Tests

One important test is that the queues' behaviour in true IPC scenarios remains correct using process forking. These tests verify that under process contention, the queues still operate correctly, properly handle shared memory regions, and that atomic operations are visible between processes. The IPC tests followed a consistent pattern using `fork()` to create separate processes as seen in listing 6.23:

Listing 6.23: IPC test structure

```
match unsafe { fork() } {
    Ok(ForkResult::Child) => {
        // Producer process
        for i in 0..NUM_ITEMS {
            loop {
                match queue.push(i) {
                    Ok(_) => break,
                    Err(_) => thread::yield_now(),
                }
            }
        }
        unsafe { libc::_exit(0) };
    }
    Ok(ForkResult::Parent { child }) => {
        // Consumer process
        let mut received = Vec::new();
        while received.len() < NUM_ITEMS {
            match queue.pop() {
                Ok(item) => received.push(item),
                Err(_) => thread::yield_now(),
            }
        }
        // Verify all items received in order
        waitpid(child, None).expect("waitpid␣failed");
    }
}
```

Line 2 in listing 6.23 creates a child process that shares the queue's memory with the parent. Lines 5 to 10 test the producer's retry loop, yielding on failure in line 8. Line 12 uses `_exit(0)` to avoid cleanup handlers that might affect shared memory. Lines 17 to 21 test the consumer loop, collecting all items. Line 24 waits for the child process to complete before final verification.

6.4.6. Special Feature Validation

Algorithm-specific features required targeted validation:

Buffering Mechanisms

Queues with local buffering like BIFFQ and MultiPush were tested like in listing 6.24 for correct flush operations and visibility of buffered items:

Listing 6.24: Buffer mechanism test

```
1 // Test BiffQ flush operations
2 for i in 0..10 {
3     queue.push(i).unwrap();
4 }
5
6 // Verify items not visible before flush
7 assert_eq!(queue.cons.shared_count.load(Ordering::Acquire), 0);
8
9 // Flush and verify visibility
10 let flushed = queue.flush_producer_buffer().unwrap();
11 assert!(flushed > 0);
12
13 // Now items should be visible to consumer
14 for i in 0..10 {
15     assert_eq!(queue.pop().unwrap(), i);
16 }
17
18 // Test automatic flush on buffer overflow
19 for i in 0..32 {  // Local buffer size
20     queue.push(i).unwrap();
21 }
22 // Should auto-flush when buffer full
23 assert_eq!(queue.local_count.load(Ordering::Relaxed), 0);
```

Lines 2 to 4 in listing 6.24 push items into the local buffer. Line 7 tests that items remain invisible to consumers before flushing. Lines 10 and 11 test explicit buffer flushing. Lines 14 to 16 test that items are now consumable in the correct order. Lines 19 to 23 test automatic flushing when the 32-item buffer fills.

Helper Thread Coordination

Verma's queue from section 5.2.4 with its helper thread requires special tests seen in listing 6.25 to verify that the helper thread functions correctly:

Listing 6.25: Helper thread coordination test

```
1  match unsafe { fork() } {
2      Ok(ForkResult::Child) => {
3          // Child runs helper thread
4          queue.run_helper();
5          std::process::exit(0);
6      }
7      Ok(ForkResult::Parent { child }) => {
8          thread::sleep(Duration::from_millis(20));
9
10         // Test operations with helper
11         assert!(queue.push(42, 0).is_ok());
12         thread::sleep(Duration::from_millis(10));
13
14         match queue.pop(0) {
15             Ok(val) => assert_eq!(val, 42),
16             Err(_) => panic!("Pop␣should␣succeed␣with␣helper"),
17         }
18
19         // Stop helper and verify cleanup
20         queue.stop_helper();
21         waitpid(child, None).unwrap();
22     }
23 }
24
25 // Test without helper - queue should still be functional
26 assert!(queue.is_empty(), "Queue␣operational␣without␣helper");
```

Line 4 in listing 6.25 starts the helper thread in a separate process. Line 8 allows time for helper initialisation. Lines 11 to 17 test that operations complete successfully with helper assistance. Line 12 provides time for the helper to process the request. Lines 20 and 21 test proper helper termination and cleanup. Finally, line 25 tests that the queue remains functional without a helper thread.

6.4.7. Miri Validation

Miri tests provided validation of memory safety in concurrent scenarios. In the test folder of the repository there are miri test files additionally to the unit test files. While structurally similar to the regular unit tests, Miri tests had to use significantly reduced parameters and simplified concurrency patterns to accommodate Miri's execution overhead. The advantage of Miri tests is that Miri simulates extreme contention leading to detect undefined behaviour easier. Miri tests will fail if undefined behaviour like data races happens.

6.4.8. Test Coverage

As we can see in fig. 6.1, a total function coverage of 81.12% and total line coverage of 70.03% was achieved, showing that most of the code paths were executed during testing. Branch coverage only lies at 44.78%, which is still fine in concurrent programming.

Figure 6.1.: Total Coverage of the Rust implementation

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
mpmc/feldman_dechev_queue.rs	82.93% (34/41)	66.56% (404/607)	67.20% (635/945)	38.97% (53/136)
mpmc/kogan_petrank.rs	83.87% (26/31)	90.44% (350/387)	92.91% (577/621)	81.71% (67/82)
mpmc/turn_queue.rs	80.00% (16/20)	83.53% (284/340)	83.41% (513/615)	58.82% (40/68)
mpmc/verma_wf.rs	80.95% (17/21)	82.30% (200/243)	78.81% (331/420)	62.50% (20/32)
mpmc/wcq_queue.rs	65.22% (30/46)	42.57% (602/1414)	40.55% (912/2249)	21.69% (82/378)
mpmc/ymc_queue.rs	81.82% (27/33)	65.28% (361/553)	63.58% (611/961)	30.00% (36/120)
mpsc/dqueue.rs	94.74% (18/19)	73.70% (426/578)	78.44% (753/960)	52.44% (86/164)
mpsc/drescher_queue.rs	100.00% (12/12)	97.39% (112/115)	98.60% (211/214)	80.00% (8/10)
mpsc/jayanti_petrovic_queue.rs	86.49% (32/37)	84.98% (345/406)	88.91% (489/550)	57.69% (30/52)
mpsc/jiffy_queue.rs	90.91% (20/22)	63.45% (486/766)	65.48% (715/1092)	40.91% (90/220)
mpsc/sesd_jp_queue.rs	100.00% (6/6)	90.67% (68/75)	93.13% (122/131)	66.67% (8/12)
spmc/david_queue.rs	70.00% (14/20)	55.69% (137/246)	56.08% (212/378)	29.17% (7/24)
spsc/biffq.rs	88.89% (16/18)	85.89% (207/241)	84.81% (335/395)	76.32% (29/38)
spsc/blq.rs	85.71% (12/14)	78.05% (128/164)	77.63% (177/228)	62.50% (10/16)
spsc/bqueue.rs	64.29% (9/14)	87.59% (120/137)	88.70% (212/239)	91.67% (22/24)
spsc/dspsc.rs	77.78% (7/9)	92.45% (98/106)	93.78% (181/193)	50.00% (4/8)
spsc/ffq.rs	81.82% (9/11)	87.76% (86/98)	86.74% (157/181)	66.67% (8/12)
spsc/iffq.rs	88.24% (15/17)	87.43% (153/175)	86.64% (253/292)	81.82% (18/22)
spsc/lamport.rs	69.23% (9/13)	81.93% (68/83)	80.65% (125/155)	100.00% (4/4)
spsc/llq.rs	80.00% (8/10)	83.33% (95/114)	85.08% (154/181)	100.00% (8/8)
spsc/mspsc.rs	69.23% (9/13)	82.05% (96/117)	83.64% (184/220)	66.67% (8/12)
spsc/sesd_jp_spsc_wrapper.rs	87.50% (7/8)	92.71% (89/96)	91.37% (127/139)	75.00% (9/12)
spsc/uspsc.rs	80.00% (8/10)	91.16% (134/147)	93.26% (249/267)	65.00% (13/20)
Totals	**81.12% (361/445)**	**70.05% (5049/7208)**	**70.83% (8235/11626)**	**44.78% (660/1474)**

7. Benchmarking and Results

To complete the objective of this thesis, finally a setting needs to be built where these algorithms are used for IPC over shared memory. This setting was also used to benchmark the performance of these algorithms to identify the best performing wait-free algorithms that can be used. In the [47] the folder benches includes the setting and benchmark of these algorithms. As seen before, these algorithms were divided into 4 categories, MPMC, MPSC, SPMC and SPSC. Therefore, 4 different IPC over shared memory settings were built to analyse firstly if all algorithms work as intended and secondly to compare the performance of all algorithms. Even though the SPMC category only has one algorithm, the built setting was still interesting to check and validate if the algorithm works as intended. After identifying the best algorithm, 3 more settings were built. One setting was to see if the SPMC queue or the best performing queues of the other categories would be faster for the SPSC category. The same was done for the MPSC category by checking if the best MPMC queue or the best MPSC queue would perform better. Finally, for the SPMC category, the same was done to check if the best MPMC or SPMC queue would be better. The best SPSC queue could not be tested for higher producer or consumer numbers, since the missing helping structures and missing atomic primitives would lead to deadlocks or inconsistent data. The benchmarks were done on a system with an Intel i7-12700H x86 processor with 14 cores. The benchmarks were implemented with the help of the `criterion` crate, which is a benchmarking library for Rust. This chapter will show in general how the benchmark settings were implemented to understand the results later.

7.1. Benchmark Structure

All benchmarks follow a consistent architecture to ensure fair comparison between queue implementations. The benchmarks measure the time taken for producers and consumers to exchange a fixed number of items through each queue implementation using IPC over shared memory. This section explains the general benchmark structure using the MPMC benchmark as a representative example, as the same patterns apply to all queue categories. Not all lines will be explained, but the most important lines will be explained to understand the general structure of the benchmarks.

7.1.1. Benchmark Parameters and Configuration

Before examining the benchmark implementation details, it is important to understand the benchmark parameters that control the test scenarios. These constants define the scale and scope of the performance measurements, as shown in listing 7.1.

Listing 7.1: Benchmark configuration constants

```
1 const ITEMS_PER_PROCESS_TARGET: usize = 170_000;
2 const PROCESS_COUNTS_TO_TEST: &[(usize, usize)] = &[(1, 1), (2, 2), (4, 4), (6, 6)];
3 const MAX_BENCH_SPIN_RETRY_ATTEMPTS: usize = 100_000_000_000;
```

Line 1 sets the number of items each producer process will generate. The value of 170,000 items provides sufficient workload to measure performance while keeping individual benchmark runs reasonable in duration. Line 2 defines the producer and consumer configurations to test as tuples. The array `[(1, 1), (2, 2), (4, 4), (6, 6)]` tests symmetric configurations from a single producer and single consumer up to 6 producers and 6 consumers, allowing analysis of scalability. Line 3 sets the maximum spin attempts before considering an operation failed. This large value ensures that wait-free operations have sufficient opportunity to complete.

For other benchmark categories, these constants are adjusted appropriately. For example, the MPSC benchmark uses:

Listing 7.2: MPSC-specific configuration

```
const ITEMS_PER_PRODUCER_TARGET: usize = 500_000;
const PRODUCER_COUNTS_TO_TEST: &[usize] = &[1, 2, 4, 8, 14];
```

The MPSC configuration tests up to 14 producers with a single consumer, using more items per producer to ensure the consumer remains busy throughout the benchmark. Similarly, SPMC and SPSC benchmarks have their own tailored parameters to effectively measure their specific use cases.

7.1.2. Benchmark Interface Implementation

Each queue implementation must provide a uniform interface for benchmarking. This is achieved through a common trait that abstracts the queue-specific operations, as shown in listing 7.3. The following pop and push traits implement the dequeue and enqueue operations respectively for the queues.

Listing 7.3: Benchmark trait for MPMC queues

```
trait BenchMpmcQueue<T: Send + Clone>: Send + Sync + 'static {
    fn bench_push(&self, item: T, process_id: usize) -> Result<(), ()>;
    fn bench_pop(&self, process_id: usize) -> Result<T, ()>;
    fn bench_is_empty(&self) -> bool;
    fn bench_is_full(&self) -> bool;
}

// Example implementation for YangCrummeyQueue
impl<T: Send + Clone + 'static> BenchMpmcQueue<T> for YangCrummeyQueue<T> {
    fn bench_push(&self, item: T, process_id: usize) -> Result<(), ()> {
        self.enqueue(process_id, item)
    }
```

```
13
14      fn bench_pop(&self, process_id: usize) -> Result<T, ()> {
15          self.dequeue(process_id)
16      }
17
18      fn bench_is_empty(&self) -> bool {
19          self.is_empty()
20      }
21
22      fn bench_is_full(&self) -> bool {
23          false  // YangCrummeyQueue has unbounded capacity
24      }
25  }
```

The trait in lines 1 to 6 defines a common interface that all MPMC queues must implement. The `process_id` parameter in lines 2 and 3 allows queues to distinguish between different processes, which is necessary for some algorithms. Lines 9 to 24 show how the YMC queue maps its specific methods to the common interface. The `bench_is_full` method in line 23 returns false for queues with unbounded capacity. This mapping is done for all queues in the respective categories, so that the benchmarks can be run with all queues without changing the benchmark code to compare fairly.

7.1.3. Process Synchronisation Infrastructure

Benchmarking concurrent algorithms requires careful synchronisation to ensure all processes start simultaneously and coordinate their completion for fair comparisons. Two synchronisation structures manage this coordination, as shown in listing 7.4.

Listing 7.4: Process synchronisation structures

```
#[repr(C)]
struct MpmcStartupSync {
    producers_ready: AtomicU32,
    consumers_ready: AtomicU32,
    go_signal: AtomicBool,
}

impl MpmcStartupSync {
    fn new_in_shm(mem_ptr: *mut u8) -> &'static Self {
        let sync_ptr = mem_ptr as *mut Self;
        unsafe {
            ptr::write(
                sync_ptr,
                Self {
                    producers_ready: AtomicU32::new(0),
                    consumers_ready: AtomicU32::new(0),
                    go_signal: AtomicBool::new(false),
                },
            );
            &*sync_ptr
```

```
21             }
22         }
23 
24         fn shared_size() -> usize {
25             std::mem::size_of::<Self>()
26         }
27 }
28 
29 #[repr(C)]
30 struct MpmcDoneSync {
31     producers_done: AtomicU32,
32     consumers_done: AtomicU32,
33     total_consumed: AtomicUsize,
34 }
```

The `MpmcStartupSync` structure in lines 1 to 6 coordinates the startup phase. Producers increment `producers_ready` in line 3 when ready, consumers increment `consumers_ready` in line 4, and all processes wait for `go_signal` in line 5 before starting. The `new_in_shm` method in lines 9 to 22 initialises the structure directly in shared memory using placement new. The `MpmcDoneSync` structure in lines 29 to 34 tracks completion, with `total_consumed` in line 33 which is used to verify that no items were lost during the benchmark.

7.1.4. Benchmark Execution Framework

The core benchmark logic is implemented in a generic function that handles process creation, execution, and measurement, as demonstrated in listing 7.5.

Listing 7.5: Generic benchmark execution function

```
fn fork_and_run_mpmc_with_helper<Q, F>(
    queue_init_fn: F,
    num_producers: usize,
    num_consumers: usize,
    items_per_process: usize,
    needs_helper: bool,
) -> Duration
where
    Q: BenchMpmcQueue<usize> + 'static,
    F: FnOnce() -> (&'static Q, *mut u8, usize),
{
    let total_items = num_producers * items_per_process;

    // Initialise queue in shared memory
    let (q, q_shm_ptr, q_shm_size) = queue_init_fn();

    // Allocate synchronisation structures
    let startup_sync_size = MpmcStartupSync::shared_size();
    let startup_sync_shm_ptr = unsafe { map_shared(startup_sync_size) };
    let startup_sync = MpmcStartupSync::new_in_shm(startup_sync_shm_ptr);
```

```rust

    let mut producer_pids = Vec::with_capacity(num_producers);
    let mut consumer_pids = Vec::with_capacity(num_consumers);

    // Fork producer processes
    for producer_id in 0..num_producers {
        match unsafe { fork() } {
            Ok(ForkResult::Child) => {
                // Signal ready and wait for go signal
                startup_sync.producers_ready.fetch_add(1, Ordering::AcqRel);
                while !startup_sync.go_signal.load(Ordering::Acquire) {
                    std::hint::spin_loop();
                }

                // Produce items
                for i in 0..items_per_process {
                    let item_value = producer_id * items_per_process + i;
                    while q.bench_push(item_value, producer_id).is_err() {
                        std::hint::spin_loop();
                    }
                }

                unsafe { libc::_exit(0) };
            }
            Ok(ForkResult::Parent { child }) => {
                producer_pids.push(child);
            }
            Err(e) => panic!("Fork failed: {}", e),
        }
    }

    // Wait for all processes to be ready
    while startup_sync.producers_ready.load(Ordering::Acquire) < num_producers as u32
        || startup_sync.consumers_ready.load(Ordering::Acquire) < num_consumers as u32
    {
        std::hint::spin_loop();
    }

    // Start timing and signal processes to begin
    let start_time = std::time::Instant::now();
    startup_sync.go_signal.store(true, Ordering::Release);

    // Wait for completion
    for pid in producer_pids {
        waitpid(pid, None).expect("waitpid failed");
    }

    start_time.elapsed()
```

```
69 }
```

The function signature in lines 1 to 10 accepts a queue initialisation function and benchmark parameters. The `needs_helper` parameter in line 6 supports queues like Verma's that require a helper thread. Line 15 initialises the queue using the provided function, which returns the queue reference and shared memory details. Lines 26 to 50 show the producer process creation where line 30 signals readiness, lines 31 to 33 implement ensures that every process waits for the go signal, and lines 36 to 41 produce items with retry logic. Lines 53 to 57 ensure all processes are ready before starting. Line 60 captures the start time immediately before signalling processes to begin in line 61. Lines 64 to 66 wait for all producer processes to complete before calculating the elapsed time in line 68.

7.1.5. Queue-Specific Benchmark Integration

Each queue type requires a specific benchmark function that integrates with the Criterion framework, as shown in listing 7.6.

Listing 7.6: Queue-specific benchmark function

```rust
fn bench_yang_crummey(c: &mut Criterion) {
    let mut group = c.benchmark_group("YangCrummeyMPMC_");

    for &(num_prods, num_cons) in PROCESS_COUNTS_TO_TEST {
        let items_per_process = ITEMS_PER_PROCESS_TARGET;
        let total_processes = num_prods + num_cons;

        group.bench_function(
            format!("{}P_{}C", num_prods, num_cons),
            |b: &mut Bencher| {
                b.iter_custom(|_iters| {
                    fork_and_run_mpmc_with_helper::<YangCrummeyQueue<usize>, _>(
                        || {
                            // Calculate required shared memory size
                            let bytes = YangCrummeyQueue::<usize>::shared_size(total_processes);
                            let shm_ptr = unsafe { map_shared(bytes) };

                            // Initialise queue in shared memory
                            let q = unsafe {
                                YangCrummeyQueue::init_in_shared(shm_ptr, total_processes)
                            };

                            (q, shm_ptr, bytes)
                        },
                        num_prods,
                        num_cons,
                        items_per_process,
                        false,  // YangCrummey doesn't need helper
```

```
29                         )
30                     })
31                 },
32             );
33         }
34 
35     group.finish();
36 }
```

Line 2 creates a benchmark group with a descriptive name. Line 4 iterates through different producer/consumer configurations from the constant array `PROCESS_COUNTS_TO_TEST`. Line 9 formats the benchmark name to indicate the configuration. Lines 11 to 30 use Criterion's `iter_custom` method to measure custom timing, as the benchmark itself measures process execution time. The closure in lines 13 to 23 initialises the queue where line 15 calculates the exact shared memory size needed and line 16 allocates the shared memory region. After that, lines 19 to 21 initialise the queue at the allocated address.

7.1.6. Consumer Process Implementation

The consumer processes follow a similar pattern but with additional logic to handle termination and verify correctness, as shown in listing 7.7.

Listing 7.7: Consumer process implementation

```
// Fork consumer processes
for consumer_id in 0..num_consumers {
    match unsafe { fork() } {
        Ok(ForkResult::Child) => {
            startup_sync.consumers_ready.fetch_add(1, Ordering::AcqRel);

            while !startup_sync.go_signal.load(Ordering::Acquire) {
                std::hint::spin_loop();
            }

            let mut consumed_count = 0;
            let target_items = total_items / num_consumers;
            let extra_items = if consumer_id < (total_items % num_consumers) {
                1
            } else {
                0
            };
            let my_target = target_items + extra_items;

            let mut consecutive_empty_checks = 0;
            const MAX_CONSECUTIVE_EMPTY_CHECKS: usize = 40000;

            while consumed_count < my_target {
                match q.bench_pop(num_producers + consumer_id) {
                    Ok(_item) => {
```

```
26                         consumed_count += 1;
27                         consecutive_empty_checks = 0;
28                     }
29                     Err(_) => {
30                         if
   ↪ done_sync.producers_done.load(Ordering::Acquire)
31                             == num_producers as u32
32                         {
33                             consecutive_empty_checks += 1;
34 
35                             if consecutive_empty_checks >
   ↪ MAX_CONSECUTIVE_EMPTY_CHECKS {
36                                 break;  // Queue likely empty
37                             }
38                         }
39 
40                         // Backoff strategy
41                         for _ in 0..100 {
42                             std::hint::spin_loop();
43                         }
44                     }
45                 }
46             }
47 
48             done_sync
49                 .total_consumed
50                 .fetch_add(consumed_count, Ordering::AcqRel);
51             done_sync.consumers_done.fetch_add(1, Ordering::AcqRel);
52 
53             unsafe { libc::_exit(0) };
54         }
55         Ok(ForkResult::Parent { child }) => {
56             consumer_pids.push(child);
57         }
58         Err(e) => panic!("Fork␣failed␣for␣consumer:␣{}", e),
59     }
60 }
```

Lines 12 to 18 calculate each consumer's share of items, distributing any remainder among the first consumers. The main consumption loop in lines 23 to 46 implements a termination strategy where lines 25 to 27 reset the empty check counter on successful pop while lines 30 to 38 check if all producers have finished and implement a termination condition. Lines 48 to 51 atomically update the total consumed count for later verification. Line 53 uses `_exit` to avoid cleanup that might interfere with shared memory.

7.1.7. Validation

After all producer and consumer processes complete, the benchmark validates that no items were lost or double read during the concurrent operations. This validation is important for ensuring the correctness of each queue implementation under IPC scenarios, as shown in listing 7.8.

Listing 7.8: Post-benchmark validation of results

```
1  // Wait for all processes to complete
2  for pid in producer_pids {
3      waitpid(pid, None).expect("waitpid for producer failed");
4  }
5
6  for pid in consumer_pids {
7      waitpid(pid, None).expect("waitpid for consumer failed");
8  }
9
10 let duration = start_time.elapsed();
11
12 // Validate that all items were consumed
13 let total_consumed = done_sync.total_consumed.load(Ordering::Acquire);
14
15 if total_consumed != total_items {
16     eprintln!(
17         "Warning (MPMC): Total consumed {}/{} items. Q: {}, Prods: {}, Cons: {}",
18         total_consumed,
19         total_items,
20         std::any::type_name::<Q>(),
21         num_producers,
22         num_consumers
23     );
24 }
25
26 // Clean up shared memory regions
27 unsafe {
28     if !q_shm_ptr.is_null() {
29         unmap_shared(q_shm_ptr, q_shm_size);
30     }
31     unmap_shared(startup_sync_shm_ptr, startup_sync_size);
32     unmap_shared(done_sync_shm_ptr, done_sync_size);
33 }
34
35 duration
```

Lines 2 to 8 wait for all processes to complete before proceeding with validation. After that, line 13 atomically reads the total number of items consumed across all consumer processes. The validation check in lines 15 to 24 compares the consumed count against the expected total. If items are missing, line 17 prints a detailed warning that includes the actual versus expected counts in line 18, the queue type name in line 20, and the producer and consumer configuration in lines 21 and 22. This warning helps identify queue implementations that may lose items under high contention or have synchronisation issues, and to verify that the queue operates correctly under concurrent access, if the warning does not appear.

7.1.8. Benchmark Configuration

The benchmarks use Criterion's configuration options to ensure reliable measurements, as shown in listing 7.9.

Listing 7.9: Criterion benchmark configuration

```
1  fn custom_criterion() -> Criterion {
2      Criterion::default()
3          .warm_up_time(Duration::from_secs(1))
4          .measurement_time(Duration::from_secs(4200))
5          .sample_size(500)
6  }
7
8  criterion_group! {
9      name = benches;
10     config = custom_criterion();
11     targets =
12         bench_wcq_queue,
13         bench_turn_queue,
14         bench_kogan_petrank_queue,
15         bench_yang_crummey
16 }
17
18 criterion_main!(benches);
```

Line 3 sets a 1-second warm-up period to stabilise the system state. Line 4 configures 4200 seconds of measurement time per benchmark. Line 5 sets 500 samples, as each sample involves creating multiple processes and produce and consume a set number of items. Lines 8 to 16 define the benchmark group with all queue implementations to test.

The same benchmark structure is applied to all benchmarks with appropriate modifications to the number of producers and consumers. This consistent approach ensures fair comparison across all implementations while accurately measuring their performance characteristics under IPC scenarios.

7.2. Benchmark Results

The benchmark results are presented for each queue category, followed by cross-category comparisons to determine the optimal wait-free data structure for different contention scenarios. All measurements represent the mean execution time in microseconds (μs) for completing the configured workload across multiple samples. In every bench, 500 samples were taken to obtain sufficient results to compare the performance. The amount of data produced and consumed was always different for each category. The queues inside each category were always tested with the same amount of data to ensure fair comparison. The number of items used for each category is shown in the respective subsections.

7.2.1. Single Producer Single Consumer (SPSC) Queue Performance

The SPSC benchmarks evaluated 11 different queue implementations with 35,000,000 items to ensure sufficient workload for accurate measurement. As shown in table 7.1, the BLQ achieved the best performance with a mean execution time of 65,199.6 μs.

Table 7.1.: SPSC Queue Performance Results (35,000,000 items)

Queue Implementation	Mean Time (μs)	Relative Performance
BLQ	65,199.6	1.00x
IFFQ	124,149.9	1.90x
LLQ	147,800.2	2.27x
BIFFQ	159,429.6	2.45x
FFQ	203,053.4	3.11x
mSPSC	331,283.6	5.08x
uSPSC	418,840.0	6.42x
JPQ's SPSC Variant	626,541.9	9.61x
Lamport's Queue	957,312.6	14.68x
B-Queue	1,552,513.1	23.81x
dSPSC	2,413,354.7	37.02x

The results reveal that the cache-aware algorithms (BLQ, IFFQ, LLQ, BIFFQ) are all faster than the other approaches, demonstrating the importance of cache optimisation. The BLQ's superior performance can be attributed to its additional batching mechanism, which amortises synchronisation costs across multiple operations while maintaining cache locality.

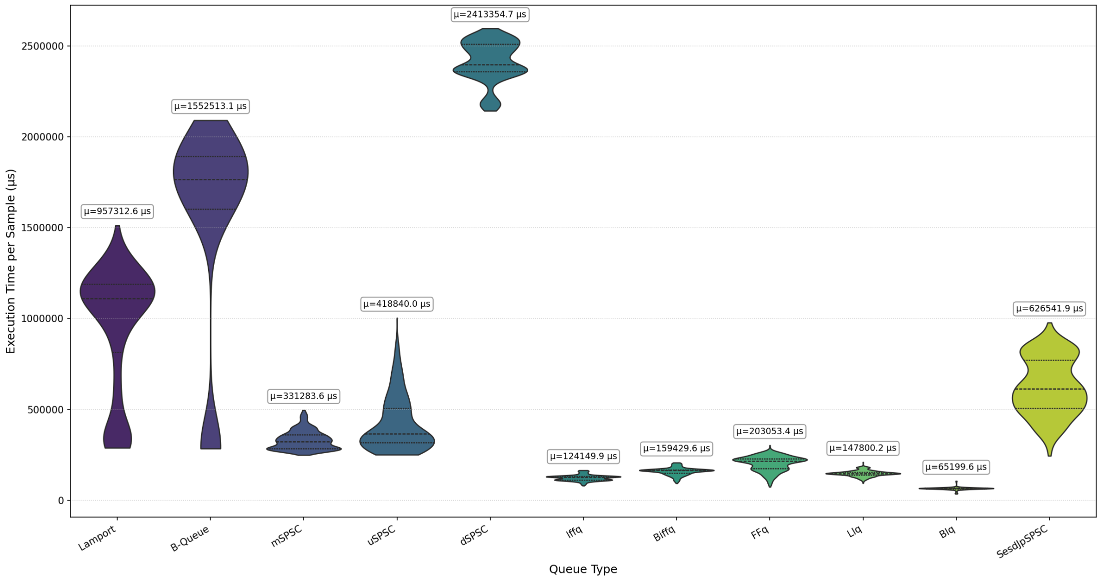
Notably, the dynamic allocation-based queues (dSPSC, uSPSC) showed worse performance, with dSPSC being 37 times slower than BLQ. This overhead stems from the memory pool management required for shared memory compatibility, as discussed in chapter 6.

The violin plot in fig. 7.1 illustrates the distribution of execution times across all SPSC implementations. It can be observed that the Lamport queue has no consistent performance with greatly varying results. It often achieves great execution times, but also often bad execution times. This is probably because of bad cache design of the queue that was talked about in algorithm 2. The BLQ on the other hand shows not only the lowest median execution time but also the most consistent performance with minimal variance, which is an important trait for designing hard timing constraints for HRTS.

7.2.2. Multi Producer Single Consumer (MPSC) Queue Performance

For MPSC scenarios, four queue implementations were tested with varying producer counts from 1 to 14, each producer generating 500,000 items. table 7.2 presents the mean performance across different producer configurations, which is visualised in fig. 7.2. DQueue with a mean

Figure 7.1.: Violin plot showing the distribution of execution times for SPSC queue implementations and 35,000,000 total items)



execution time of 13,967.2 μs for 1 producer, 29,039.9 μs for 2 producers, and scaling up to 323,126.2 μs for 14 producers, outperformed all other implementations.

Table 7.2.: MPSC Queue Performance Results (500,000 items per producer)

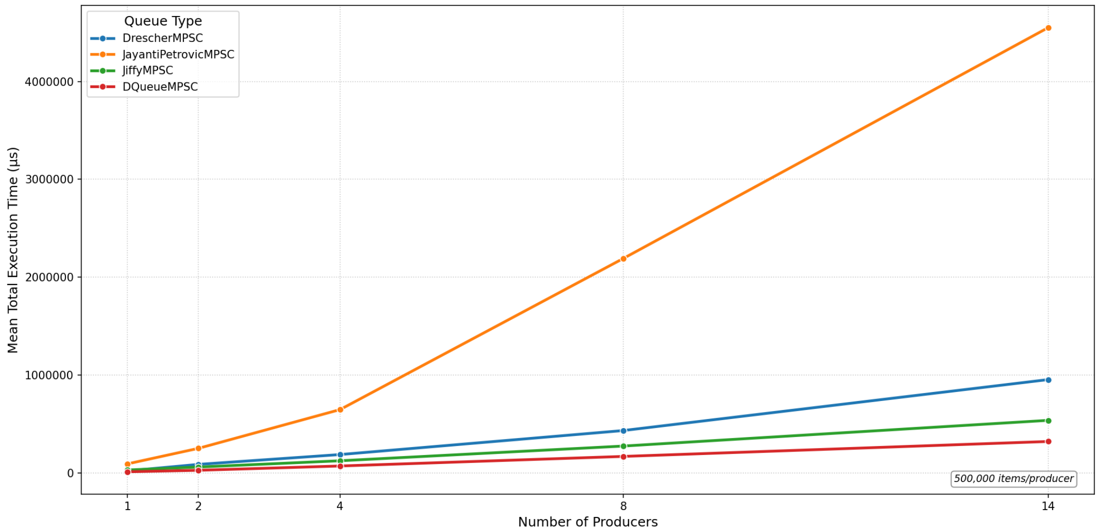
Queue	1P	2P	4P	8P	14P
DQueue	13,967.2	29,039.9	72,894.1	170,676.4	323,126.2
Drescher	22,932.0	89,353.0	189,703.0	434,820.2	955,047.8
Jiffy	32,280.3	62,159.1	126,199.3	276,401.5	538,978.2
JPQ's MPSC Variant	94,912.5	252,959.1	648,641.6	2,191,529.2	4,548,422.3

DQueue had the best performance overall as producer count increased. The DQueue's local buffering mechanism effectively reduces contention by minimising synchronisation operations, as each producer accumulates items locally before batch-writing to the shared queue.

The JPQ, despite its theoretical $O(\log n)$ complexity, showed poor practical performance. This is due to the overhead of maintaining the binary tree structure for timestamp propagation. This highlights the gap between theoretical complexity and real-world performance in concurrent data structures.

fig. 7.3 shows the performance distribution under maximum contention (14 producers). DQueue maintains a tight distribution even under high producer contention, which is also the

Figure 7.2.: Mean execution time of MPSC queue implementations as producer count increases



case for lesser producer counts as seen in figs. A.1 to A.4. This shows that DQueue's design is ensuring consistent enough performance to design hard timing constraints for HRTS.

7.2.3. Single Producer Multi Consumer (SPMC) Queue Performance

Only one native SPMC implementation was available. So no performance comparison was made with another SPMC queue. What was done is comparing the performance of the David queue with the best performing MPMC queue in a SPMC benchmark setting, to see if maybe the best performing MPMC queue would be even better in a SPMC setting than a native SPMC queue. This can be seen in section 7.2.5.

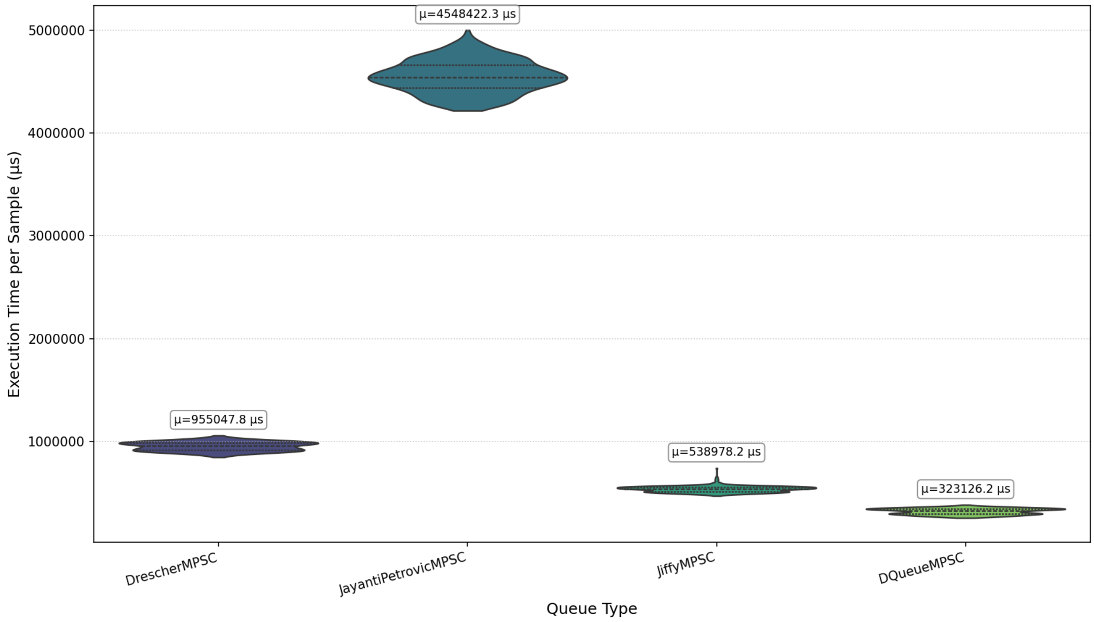
7.2.4. Multi Producer Multi Consumer (MPMC) Queue Performance

The MPMC category included six implementations tested with symmetric producer-consumer configurations. Each producer generated 170,000 items. table 7.3 shows the mean performance across different producer-consumer configurations, which is again visualised as seen in fig. 7.4.

The YMC queue achieved the best overall performance with 72,910.8 μs for 1 producer and 1 consumer, 72,547.6 μs for 2 producers and 2 consumers, and scaling to 121,477.9 μs for 6 producers and 6 consumers. Its performance remained relatively stable across different configurations, demonstrating its efficiency in handling multiple producers and consumers.

What can be observed is that the Verma queue has better performance in the 1P/1C case with a mean performance of 45,690.9 μs than YMC. The reason for that is most probably

Figure 7.3.: Violin plot showing the distribution of execution times for MPSC queue implementations with 14 producers and 1 consumer and 7,000,000 total items



because even in the 1P/1C case there is still an active dedicated helper thread that is used to help the producer and consumer processes.

fig. 7.5 illustrates the performance distribution under symmetric high contention (6P/6C). The YMC queue demonstrates that its performance stays consistent with minimal variance, even in lesser producer and consumer counts as seen in figs. A.6 and A.7, indicating predictable performance beneficial to define tight timing constraints for HRTS. The only exception is the 1P/1C case, where the performance distribution is more spread out, as seen in fig. A.5, indicating that YMC is not as optimised for low contention scenarios as it is for high contention scenarios. The Verma queue has consistent performance over all process counts, but with a higher mean execution time than YMC. So for HRTS which requires predictability, Verma queue could be a better choice even though it is not the fastest queue in the MPMC category.

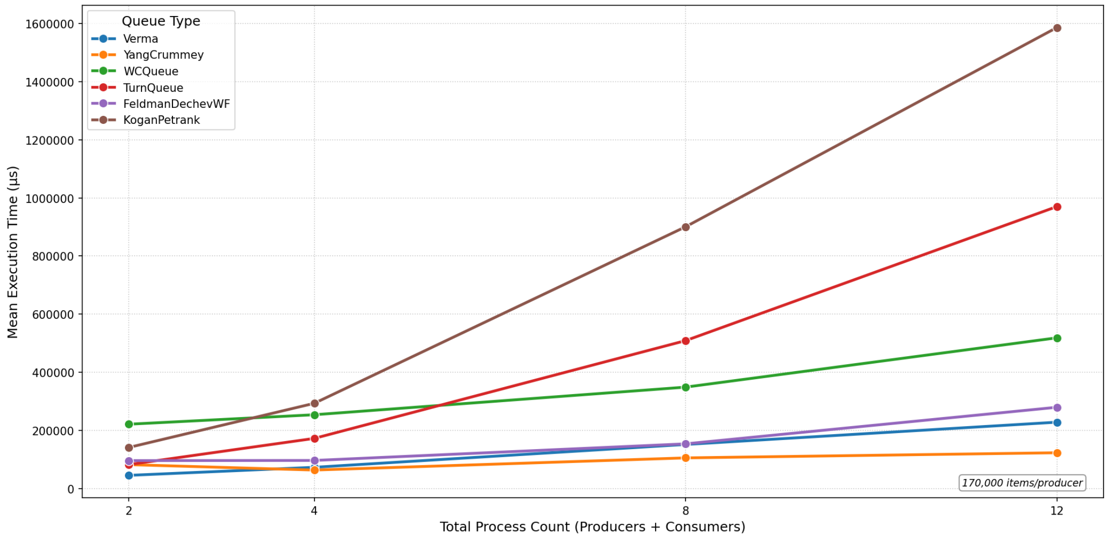
7.2.5. Cross-Category Performance Comparison

To determine whether specialised queues for each contention category are necessary, cross-category benchmarks were conducted comparing the best performers from each category against queues from other categories. The specialised queues for their respective contention scenarios were called native in each cross-category benchmark. The results are summarised in the following subsections.

Table 7.3.: MPMC Queue Performance Results (170,000 items per producer)

Queue	1P/1C	2P/2C	4P/4C	6P/6C
YMC	72,910.8	72,547.6	101,541.4	121,477.9
Verma	45,690.9	73,042.7	153,716.8	229,763.6
FeldmanDechev	90,956.0	100,893.5	150,636.2	278,037.7
TurnQueue	100,150.3	173,204.2	510,697.0	971,945.7
KoganPetrank	174,502.6	285,645.9	896,079.8	1,574,045.2
wCQ	233,996.9	248,234.3	350,401.3	518,476.6

Figure 7.4.: Mean execution time of MPMC queue implementations as process count increases



Best Queue for SPSC Scenarios

table 7.4 compares the native SPSC winner (BLQ) against the best queues from other categories operating in SPSC mode with 300,000 items.

The native SPSC queue with a mean performance of $3,625.7\mu s$ outperformed all others, being 2.11x faster than DQueue and nearly 10x faster than YMC. This demonstrates that specialised SPSC algorithms provide benefits when contention is limited to a single producer and consumer pair. fig. 7.6 visualises this again.

Best Queue for MPSC Scenarios

Comparing the native MPSC winner DQueue against YMC operating in an MPSC setting reveals similar patterns, as shown in table 7.5 visualised in fig. 7.7.

Figure 7.5.: Violin plot showing the distribution of execution times for MPMC queue implementations with 6 producers and 6 consumers and 1,020,000 total items

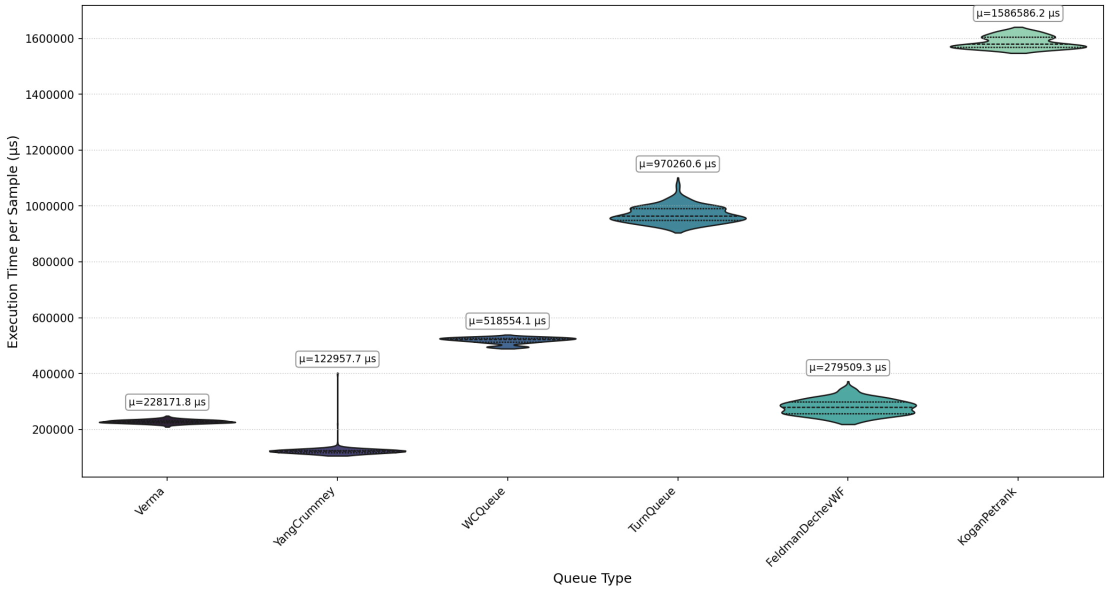


Table 7.4.: Cross-Category Performance in SPSC Configuration (300,000 items)

Queue (Category)	Mean Time (μs)	Relative to BLQ
BLQ (Native SPSC)	3,625.7	1.00x
DQueue (MPSC as SPSC)	7,638.9	2.11x
David (SPMC as SPSC)	21,207.3	5.85x
YMC (MPMC as SPSC)	36,170.9	9.98x

DQueue outperformed YMC across all producer counts, with the performance gap decreasing from 6.22x at 1 producer to 2.32x at 14 producers. The native MPSC implementation maintains its advantage due to its optimised local buffering mechanism that reduces synchronisation overhead. YMC has overhead from its additional consumer synchronisation mechanisms, which are not needed in an MPSC setting.

fig. 7.8 illustrates the performance distribution comparison between DQueue (native MPSC) and YMC (operating as MPSC) under maximum producer contention (14 producers). DQueue shows slightly better consistency whilst performing faster than YMC, showing that also in MPSC settings specialised queues are better than more general MPMC queues. The same goes for lesser producer counts as seen in figs. A.8 to A.11.

Figure 7.6.: Violin plot showing performance distribution of different queue categories operating in an SPSC setting with 300,000 total items

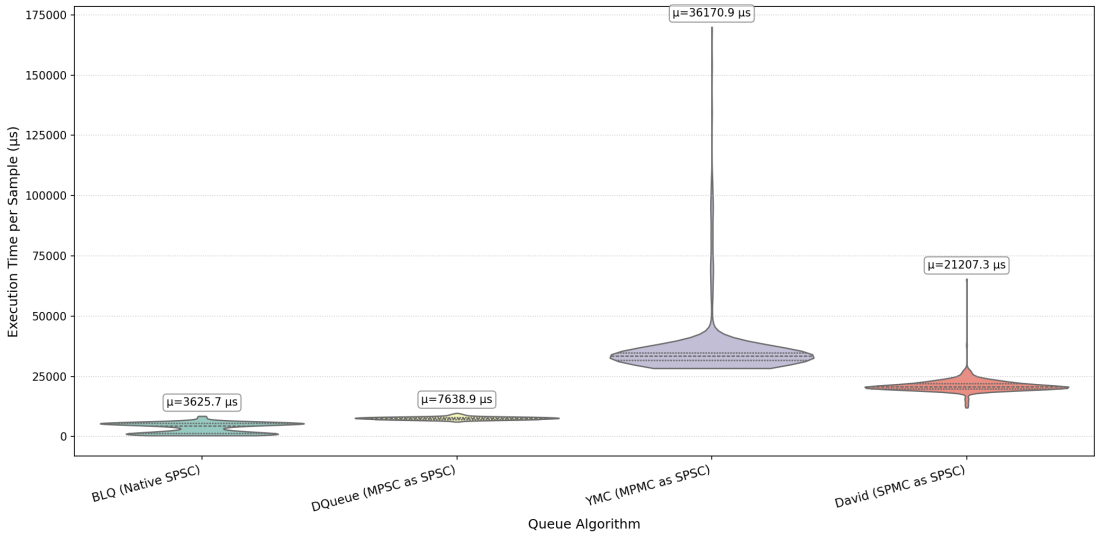


Table 7.5.: Cross-Category Performance in MPSC Configuration (100,000 items per producer)

Producers	DQueue (Native MPSC) Time (μs)	YMC (as MPSC) Time (μs)	Relative to DQueue
1	2,102.7	13,075.6	6.22x
2	7,081.9	19,650.4	2.77x
4	16,588.1	39,848.3	2.40x
8	36,197.5	84,092.5	2.32x
14	72,788.4	168,854.4	2.32x

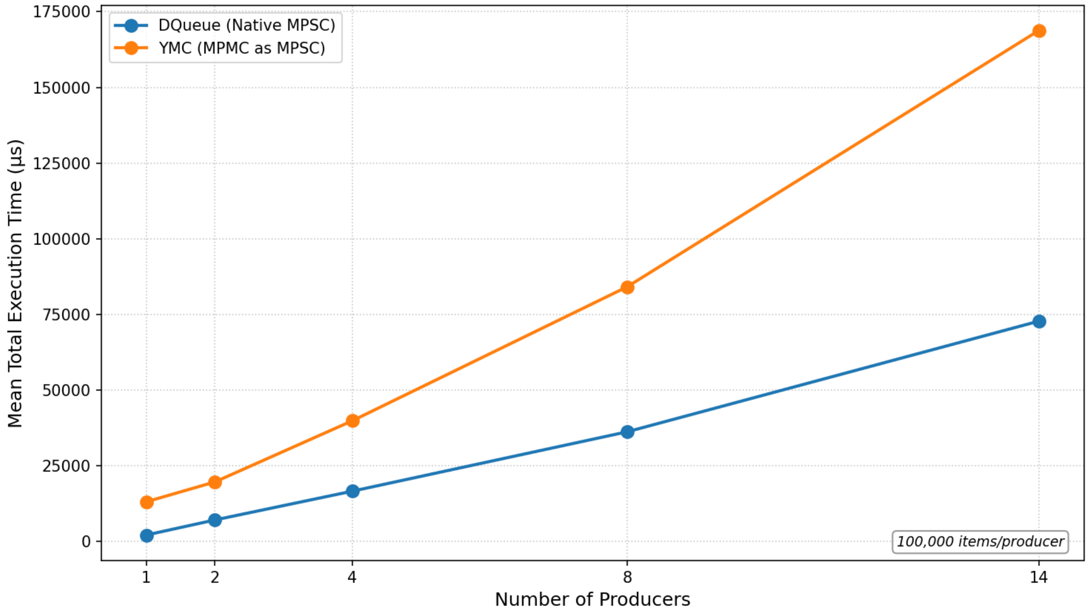
Best Queue for SPMC Scenarios

The native SPMC implementation (David) was compared against YMC in SPMC mode, as presented in table 7.6.

Similar to the MPSC results, the specialised David queue significantly outperformed YMC, maintaining a 2.5-6x performance advantage across different consumer counts. The native SPMC algorithm's two-dimensional array design with row jumping proves to be faster than the MPMC approach.

The violin plot in fig. 7.9 demonstrates the performance distribution under maximum consumer contention. The David queue shows more consistent performance with a tighter distribution whilst being faster compared to YMC operating in SPMC mode, confirming again

Figure 7.7.: Mean execution time comparison of DQueue (native MPSC) vs YMC (as MPSC) across different producer counts



specialised queues are the better choice. Also here the same can be said for lesser consumer counts as seen in figs. A.12 to A.15.

Table 7.6.: Cross-Category Performance in SPMC Configuration (100,000 items per consumer)

Consumers	David (Native SPMC) Time (μs)	YMC (as SPMC) Time (μs)	Relative to David
1	2,754.4	16,696.0	6.06x
2	8,222.4	24,220.6	2.95x
4	17,243.9	44,368.2	2.57x
8	37,756.0	96,410.1	2.55x
14	71,305.7	225,931.3	3.17x

Figure 7.8.: Violin plot showing the distribution of execution times for cross-category MPSC comparison with 14 producers and 1 consumer and 1,400,000 total items

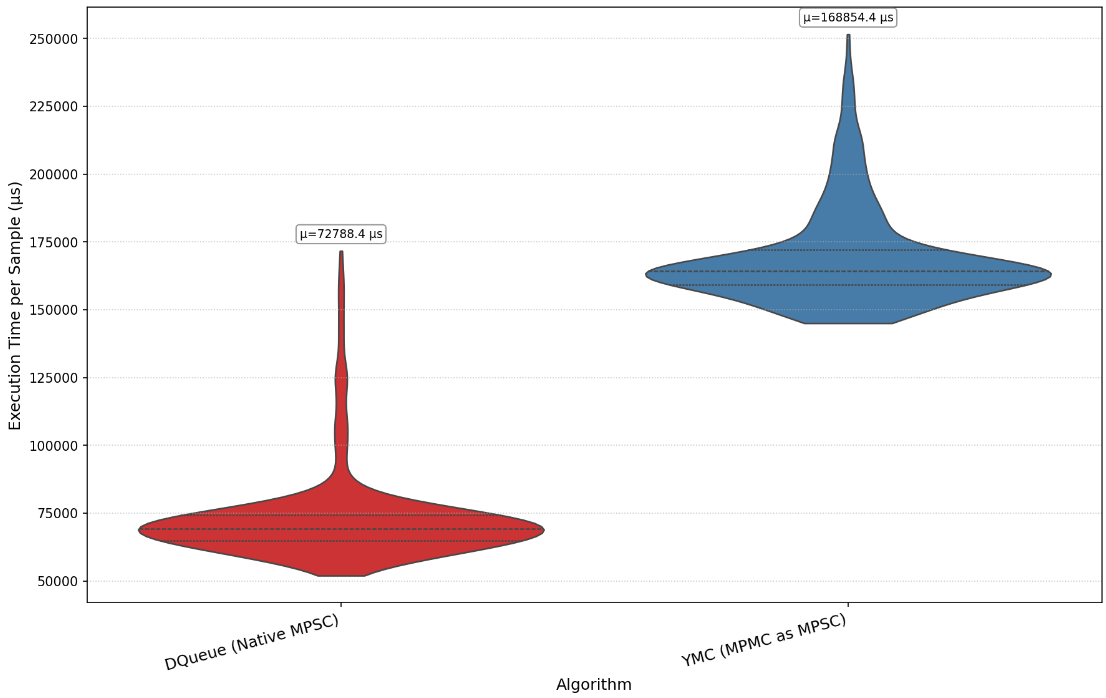


Figure 7.9.: Violin plot showing the distribution of execution times for SPMC implementations with 1 producer and 14 consumers and 1,400,000 total items

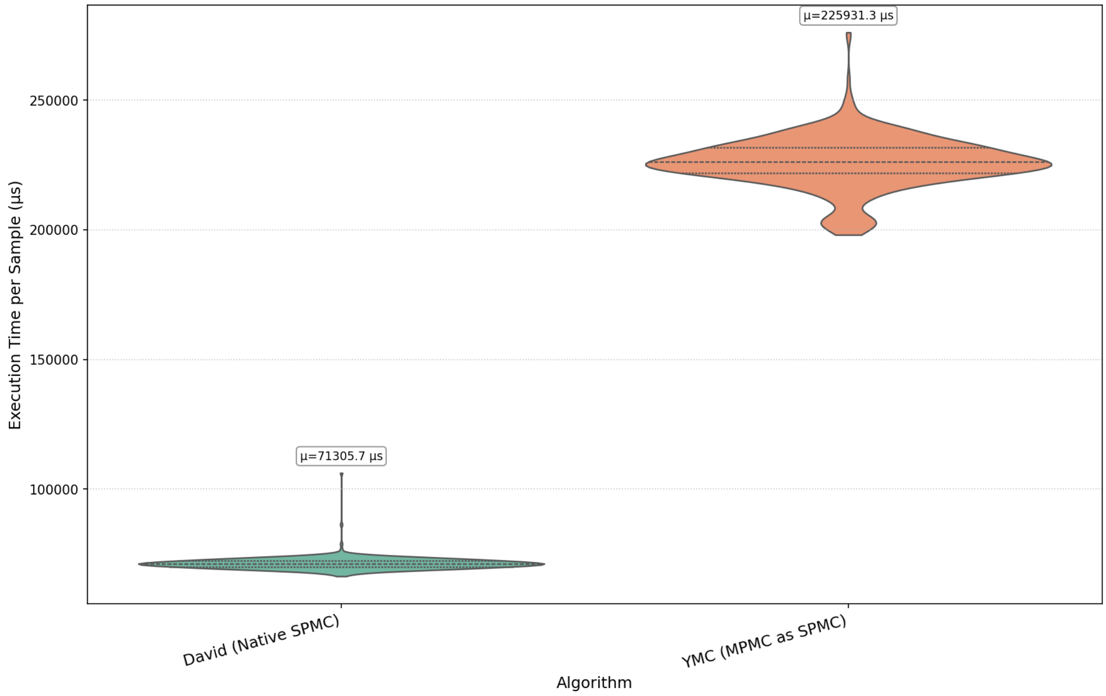
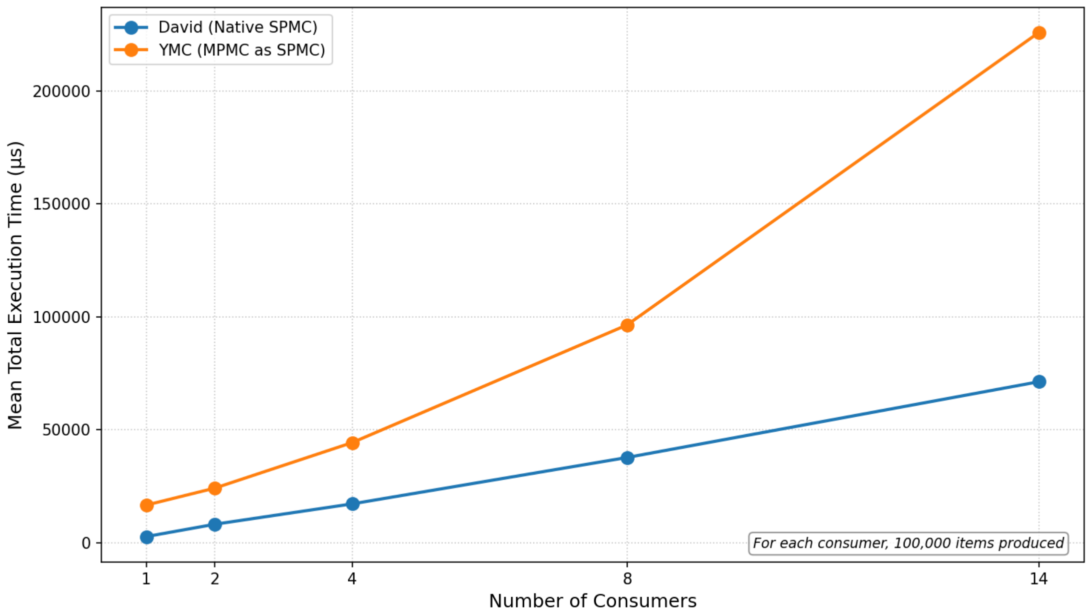


Figure 7.10.: Mean execution time comparison of David (native SPMC) vs YMC (as SPMC) across different consumer counts



8. Conclusion and Future Work

8.1. Conclusion

This thesis investigated wait-free data structures for IPC through shared memory for RTSs, with implementations in Rust. The primary goal was to identify and evaluate the best wait-free algorithms that can provide predictable timing guarantees essential for HRTS.

Through systematic analysis of 20 different wait-free queue implementations across four contention categories (SPSC, MPSC, SPMC, and MPMC), several key insights emerged. First, specialisation matters significantly in concurrent data structures. The benchmarks demonstrated that algorithms optimised for specific producer-consumer settings outperform solutions built for other contention categories by factors of 2 to 10. This performance gap justifies maintaining separate implementations for different contention scenarios rather than, for example, relying on a single MPMC queue for all use cases.

Cache optimisation and batching proved to be a critical factor for performance. The Batched Lamport Queue (BLQ) achieved the best SPSC performance at 65.2 milliseconds for 35 million items, due to its cache-aware design that minimises false sharing through cache line separation and batching through a producer that accumulates data before pushing them and making them visible for a consumer. Similarly, DQueue was the best-performing queue in the MPSC category by using batching. These results confirm that understanding and optimising cache hierarchies is more important than just proven better theoretical algorithmic time complexity, like JPQ, in practice.

The implementation challenges of adapting thread-based algorithms to IPC revealed important practical considerations. Dynamic memory allocation, which is simpler in thread-based implementations, required a significant redesign for shared memory contexts. All queues using dynamic allocation had to be modified to use pre-allocated memory pools, introducing additional complexity but ensuring position-independent operation across different process address spaces. This adaptation particularly impacted the performance of dSPSC and uSPSC queues.

The Rust programming language was well-suited for implementing these algorithms. Its ownership model and explicit memory ordering semantics allowed precise control over synchronisation while maintaining memory safety. The unsafe blocks required for shared memory operations were well-contained, and the type system helped prevent common concurrency bugs during development. The achieved test coverage of 81.12% function coverage and 70.03% line coverage demonstrates that most code paths could be reliably tested despite the complexity of concurrent operations. These tests, along with the additional correctness check in the benchmark, also provided validation that these queues work as expected.

The implemented queues were then benchmarked to answer the objective of this thesis. The benchmarks showed that the recommended wait-free queues for real-time IPC are Batched Lamport Queue (BLQ) for SPSC, DQueue for MPSC, David Queue for SPMC, and YMC or Verma Queue for MPMC depending on the level of contention.

8.2. Future Work

While this thesis provides a comprehensive evaluation of existing wait-free algorithms for IPC, several areas require further research.

8.2.1. Dynamic Wait-Free Memory Allocation for Shared Memory Inter-Process Communication (IPC)

The most problematic area during implementation was the lack of wait-free dynamic memory allocation schemes suitable for shared memory contexts. All algorithms requiring dynamic allocation had to be modified to use pre-allocated pools, which limits flexibility and potentially wastes memory. Future research should investigate wait-free memory allocators specifically designed for shared memory IPC.

Such allocators need to address the following challenge. They must operate without a global heap, as each process has its own address space. So, such an allocator would need to share the address space of the shared memory region across all processes in a wait-free manner. Also, it needs to function so that dynamically the shared memory region can shrink and grow as needed while sharing the new memory region bounds to the processes while still maintaining wait-freedom.

A wait-free shared memory allocator would enable more flexible data structures and could allow the exact implementation of dynamic queues like dSPSC and uSPSC that currently cannot be that dynamic with a pre-allocated pool.

8.2.2. Integration with Real-Time Operating System (RTOS)

The current benchmarks run on a Linux system, which introduces timing variations from scheduling and interrupts. Future work should evaluate these algorithms on RTOS like RTEMS or QNX to observe the performance in an RTS environment. This would provide insights into how these wait-free queues perform under real-time constraints and scheduling policies.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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`.

[22] 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`.

[23] 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`.

[24] 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`.

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

[26] 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`.

[27] 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.

[28] 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`.

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

[30] 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`.

[31] 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`.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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`.

[43] 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.

[44] 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`.

[45] 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`.

[46] 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`.

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

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

[49] 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`.

[50] 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`.

[51] 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-Width 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 Dynamic Single Producer Single Consumer

mSPSC MultiPush Single Producer Single Consumer

BLQ Batched Lamport Queue

LLQ Lazy Lamport Queue

wCQ Wait-Free Circular Queue

YMC Yang Mellor-Crummey

sCQ Scalable Circular Queue

JPQ Jayanti Petrovic Queue

List of Figures

List of Tables

List of Algorithms

Listings

A. Appendix

A.1. Additional Benchmark Visualisations

This section presents additional violin plots from the benchmark results, illustrating performance distributions for various producer and consumer configurations. These complement the main results presented in chapter 7.

A.1.1. MPSC Queue Performance Distributions

The following figures show the performance distribution of MPSC queue implementations across different producer configurations:

Figure A.1.: MPSC queue performance distribution with 1 producer with 500,000 Total Items

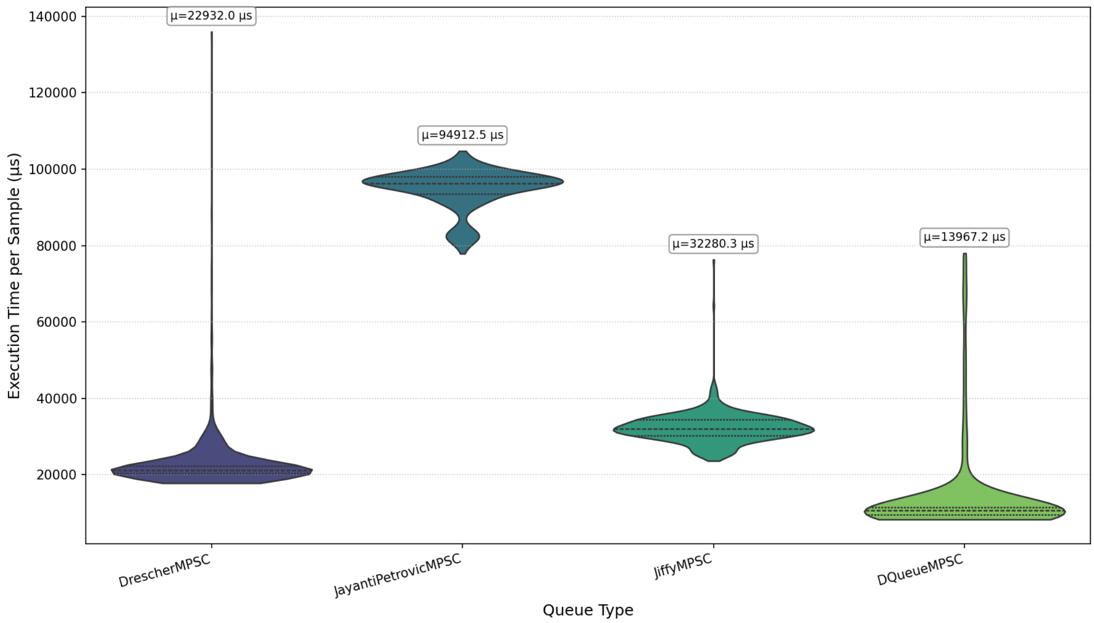


Figure A.2.: MPSC queue performance distribution with 2 producers with 1,000,000 Total Items

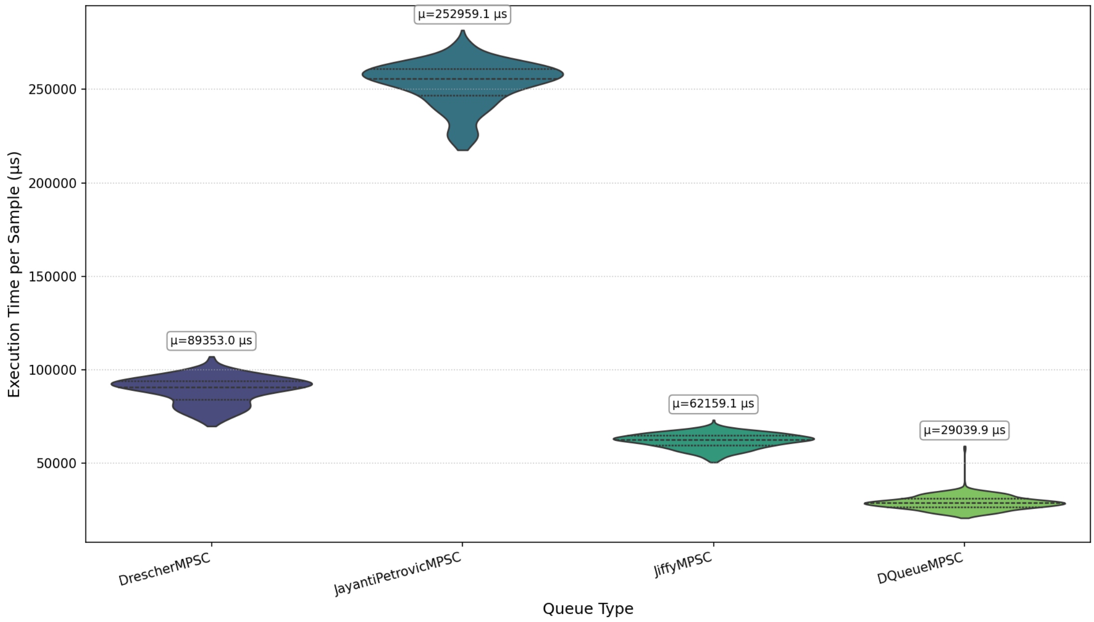


Figure A.3.: MPSC queue performance distribution with 4 producers with 2,000,000 Total Items

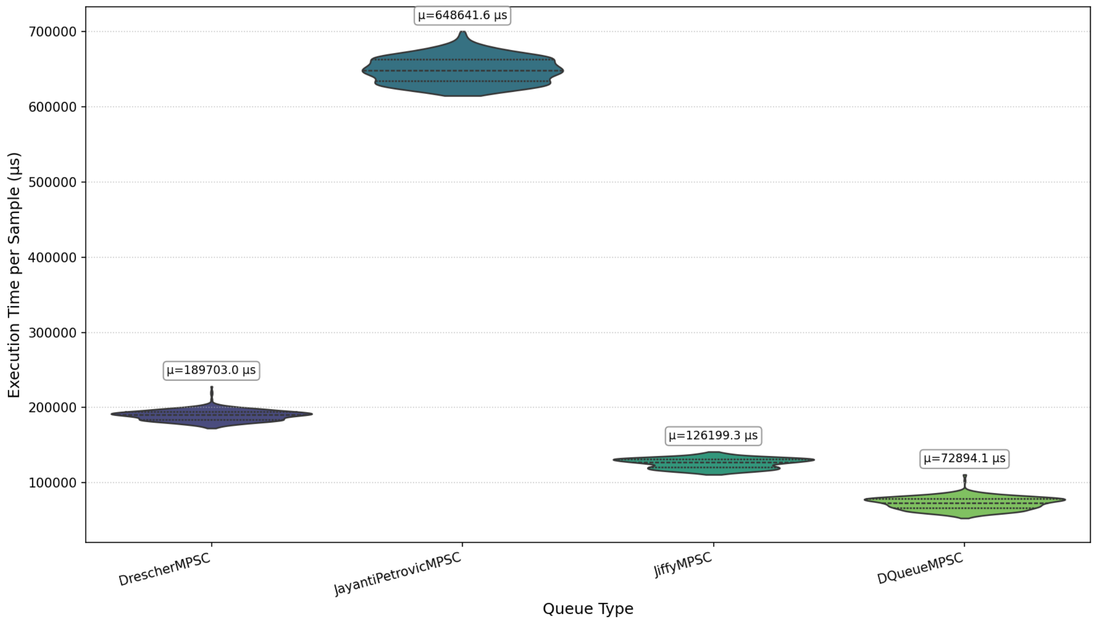
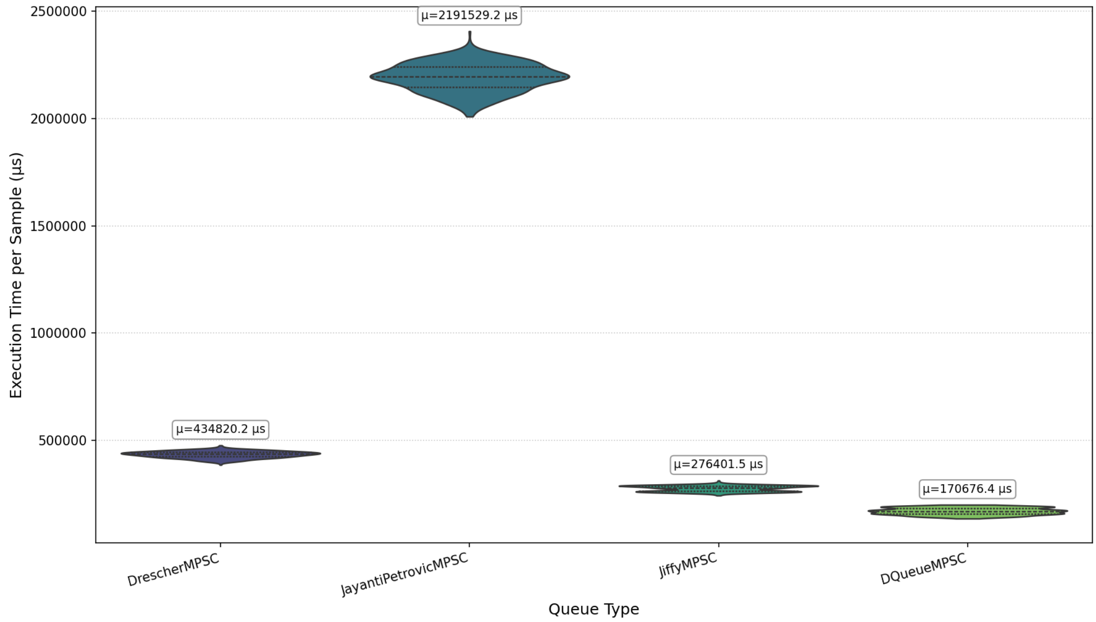


Figure A.4.: MPSC queue performance distribution with 8 producers with 4,000,000 Total Items



A.1.2. MPMC Queue Performance Distributions

The following figures show the performance distribution of MPMC queue implementations across different configurations:

Figure A.5.: MPMC queue performance distribution with 1 producer and 1 consumer with 170,000 Total Items

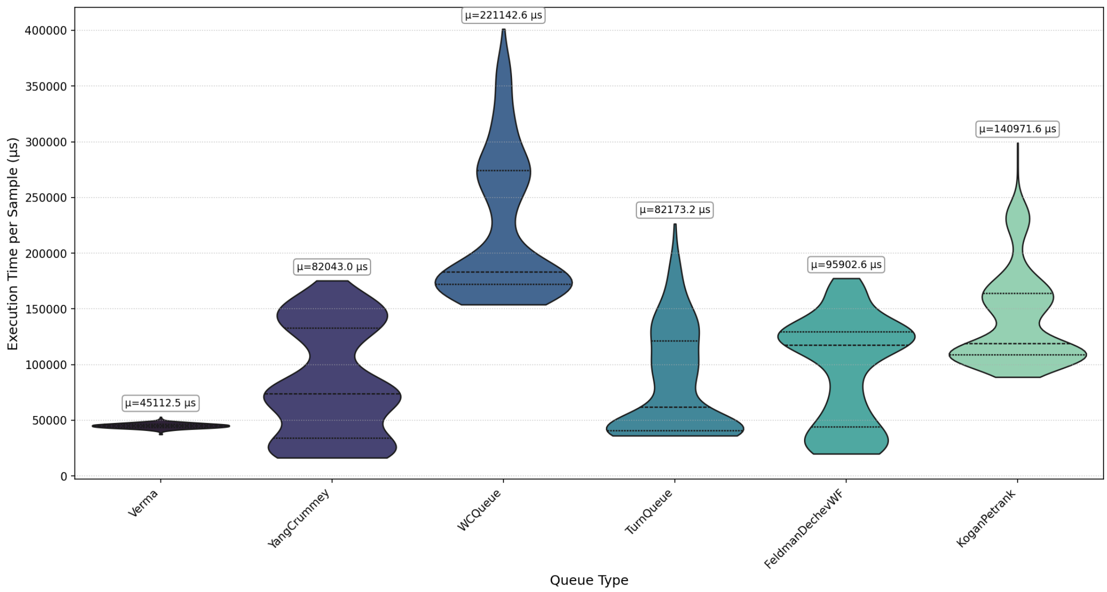


Figure A.6.: MPMC queue performance distribution with 2 producers and 2 consumers with 340,000 Total Items

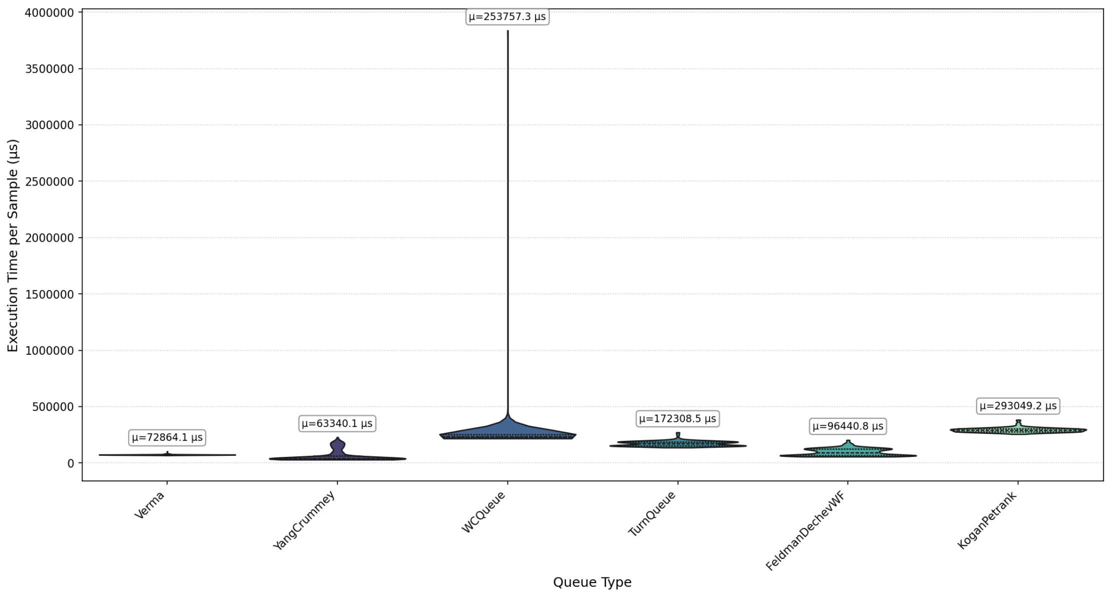
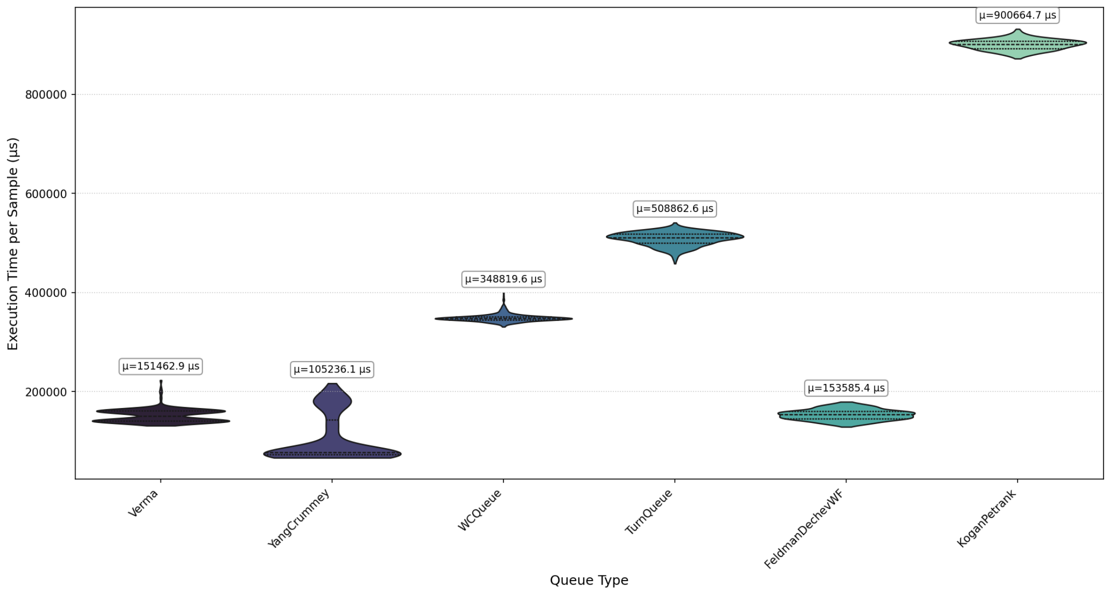


Figure A.7.: MPMC queue performance distribution with 4 producers and 4 consumers with 680,000 Total Items



A.1.3. Cross-Category Performance Distributions

The following figures show performance distributions when queues from different categories operate in various contention scenarios:

Cross-Category MPSC Performance

Figure A.8.: Cross-category MPSC performance distribution with 1 producer and 1 consumer with 100,000 Total Items

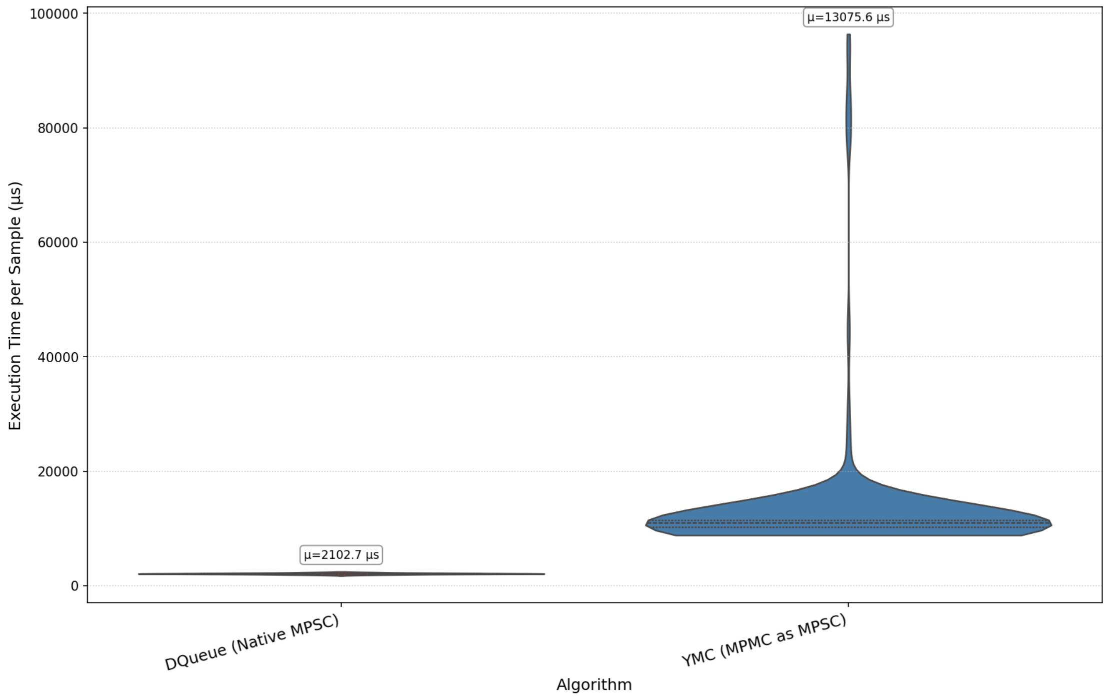


Figure A.9.: Cross-category MPSC performance distribution with 2 producers and 1 consumer with 200,000 Total Items

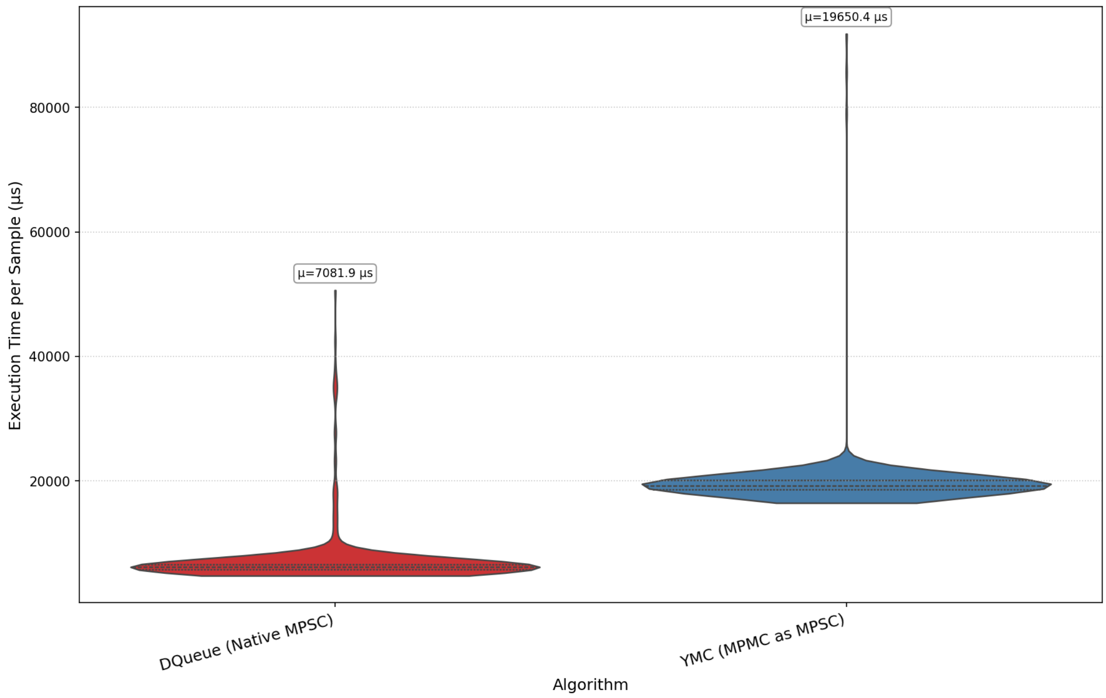


Figure A.10.: Cross-category MPSC performance distribution with 4 producers and 1 consumer with 400,000 Total Items

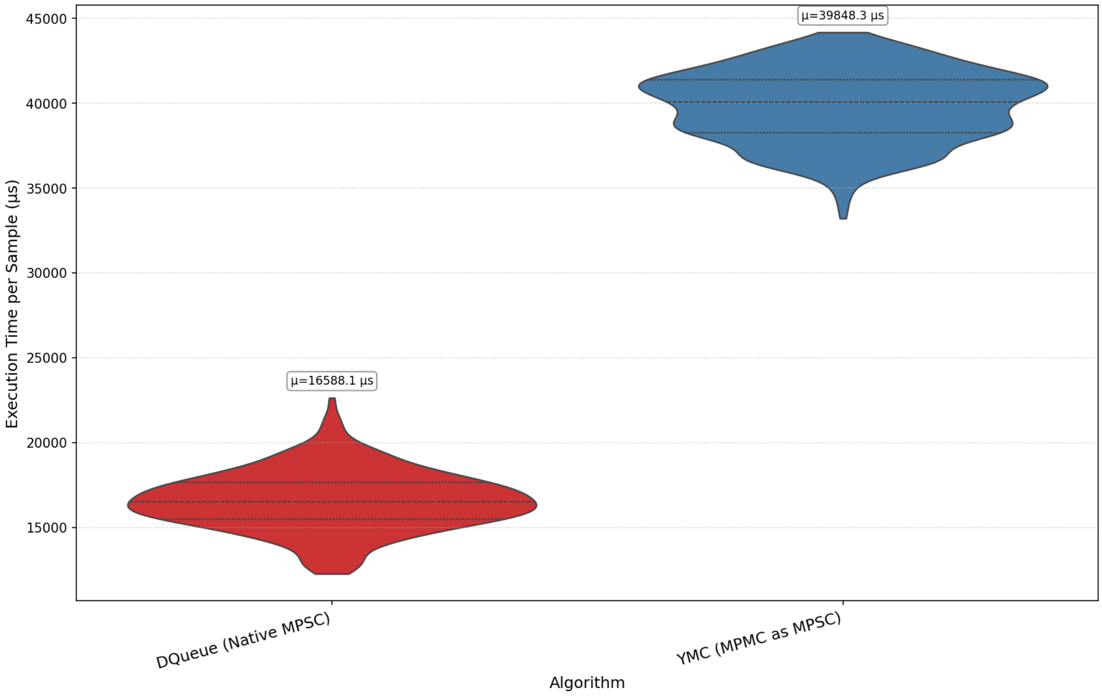
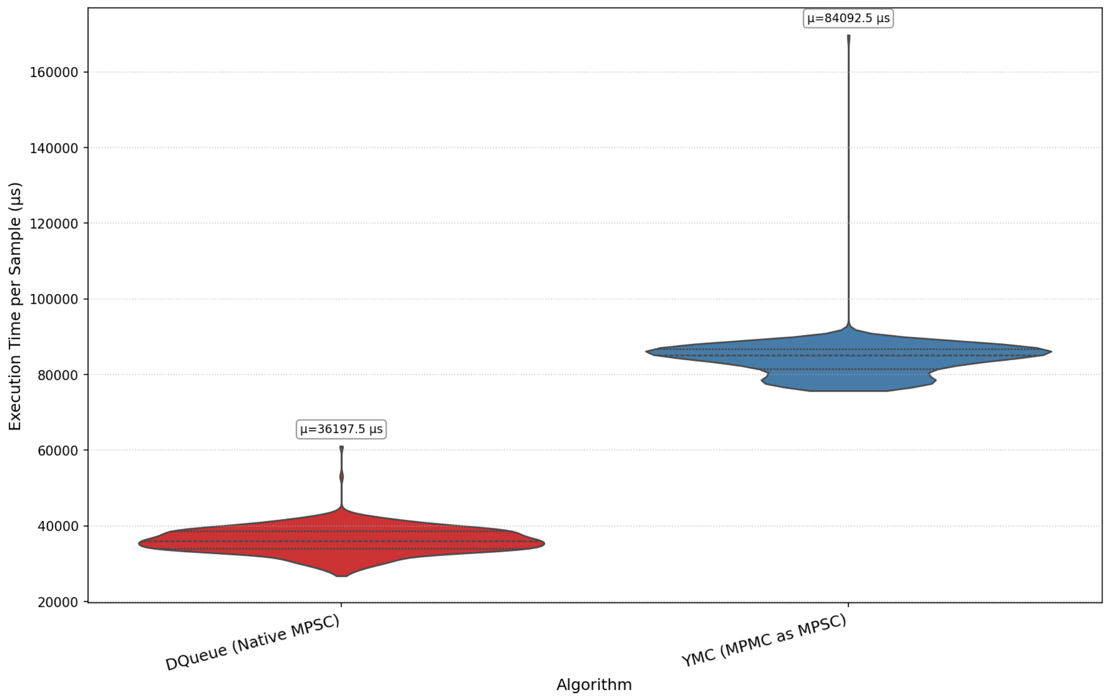


Figure A.11.: Cross-category MPSC performance distribution with 8 producers and 1 consumer with 800,000 Total Items



Cross-Category SPMC Performance

Figure A.12.: Cross-category SPMC performance distribution with 1 producer and 1 consumer with 100,000 Total Items

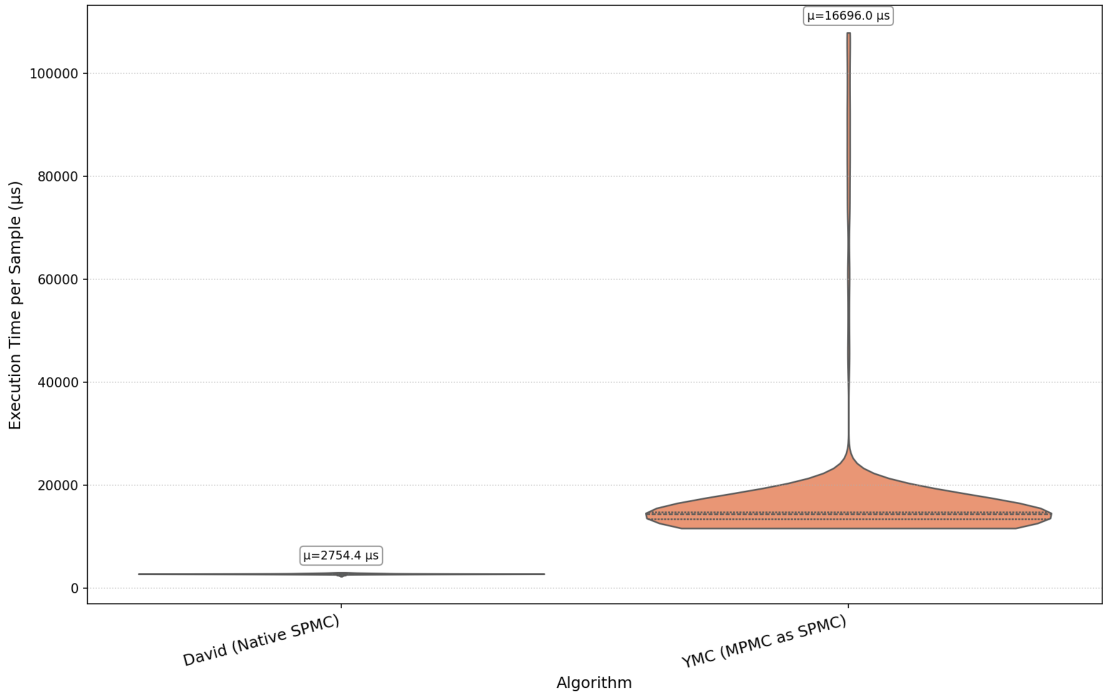


Figure A.13.: Cross-category SPMC performance distribution with 1 producer and 2 consumers with 200,000 Total Items

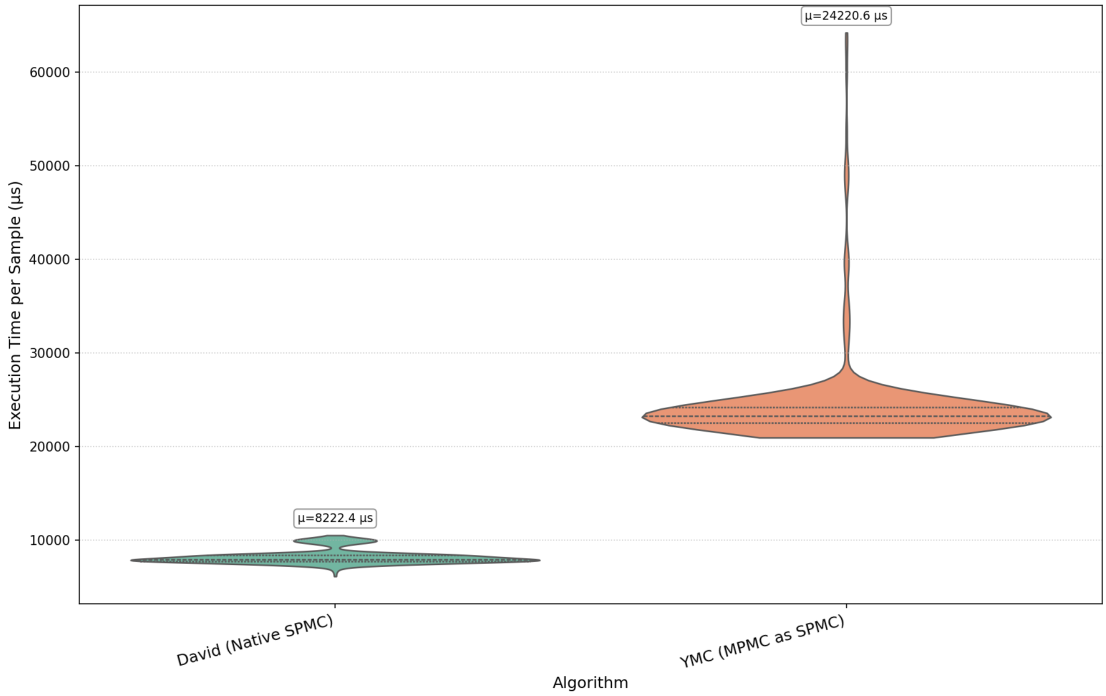


Figure A.14.: Cross-category SPMC performance distribution with 1 producer and 4 consumers with 400,000 Total Items

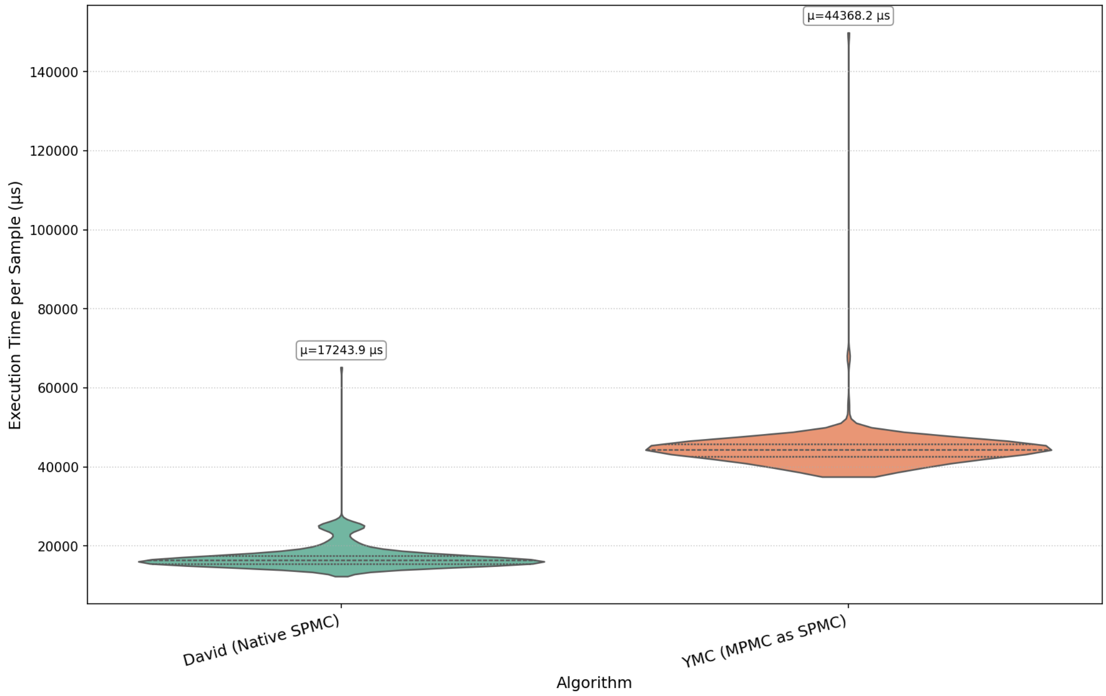


Figure A.15.: Cross-category SPMC performance distribution with 1 producer and 8 consumers with 800,000 Total Items