A Modular Framework for Benchmarking FIFO Queues

Ein modularer Framework zum Benchmarking von FIFO Queues

Jann Peter Vincent Stute

Universitätsbachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
im Studiengang
IT-Systems Engineering

Eingereicht am 30. September 2025 am
Fachgebiet für Internet-Technologien und Softwarization
der Digital Engineering-Fakultät
der Universität Potsdam

Betreuer: Prof. Dr. Holger Karl

Abstract

In implementing parallelised architectures, First In, First Out (FIFO) queues can play an important role as communication channels between different application components[1]. These communication channels can often be core to the performance of such applications, particularly network applications, which often make use of pipeline parallelism[1], [2]. However, there are many different Single Producer Single Consumer (SPSC) FIFO queue implementations, and the authors of each paper benchmark their queue differently, making it hard to compare results between them. For this reason, it is difficult to determine the optimal choice of queue for an application without implementing and testing all of the considered options. I seek to alleviate this problem by building a benchmarking framework, which should allow for easier benchmarking of a wide range of queues and help future authors of papers presenting novel queue designs to benchmark their newly proposed implementations in a replicable and comparable manner. I make use of this framework to test a range of different queues: B-Queue, EQueue, FastForward Queue, FastFlow Queue, Lamport Queue and MCRingBuffer [3], [4], [5], [6], [7], [8]. In the benchmarks used here, FastForward Queue, FastFlow Queue, and MCRingBuffer were consistently either the best or among the best-performing queues. I also compared my results with some of those of the EQueue, B-Queue, and FastForward papers and found that they are similar, but not identical. Furthermore, I published all source code used for this benchmarking, along with detailed contextual information, in order to make my results as simple to replicate as possible.

Contents

1 Introduction

Parallelisation of applications is an important technique for reaching performance goals. A common example of such a utilisation of multi-core architectures is network applications making use of pipeline parallelism[1], [2]. This allows utilising multiple threads by decomposing these network applications into sequential tasks that can be parallelised[1]. In such applications, FIFO queues play an important role as communication channels between these sequential tasks, which can also often be located on different CPU cores. However, the performance of these communication channels can be critical to effectively making use of parallelisation, "take [a] 10 Gbps network as an example, a minimal 64-byte Ethernet frame must be processed in amortized time of 67 ns, which is about one DRAM access time"[3].

Because of this, there are many different SPSC FIFO queue designs aiming to improve, relative to previous iterations, the time spent per enqueue/dequeue operation[3], [4], [5], [6], [8]. However, choosing one of these implementations can be difficult for several reasons. Each paper concludes that its proposed design surpasses the performance of existing designs in some respect and presents test results to support the performance improvement in that respect. This means that these benchmarks are very different from paper to paper, making it hard to compare the results. Furthermore, there also don't seem to be any papers comparing many different queues via the same tests (See Appendix B). The papers also rarely provide source code for their new queue designs, or, if they do, the source code may not be available anymore due to its external hosting having become unavailable.[1] All of this makes a direct replication hard, if not impossible[9].

With the aim of addressing these problems, I built a benchmarking framework and used it to benchmark six queues: B-Queue, EQueue, FastForward Queue, FastFlow Queue, Lamport Queue, and MCRingBuffer[2] [3], [4], [5], [6], [7], [8]. This Framework should make it easier for future authors to test queue designs in a way that is replicable and comparable, and make it easy to include parameter values like queue size, batching size, etc., with the source code by reading these values from a config file that can be stored alongside it. Afterwards, I compare the results from the benchmarks to those of the EQueue, B-Queue and FastForward papers in order to see how closely their results are replicated. These benchmarks were designed to be most similar to those of the EQueue paper, as that paper compares the largest number of different queues out of those three papers.

In order to make my results as replicable as possible, I released the full source code on GitHub (see Computerdores/ba) and also embedded it into the PDF to prevent it from becoming unavailable in the future (see Appendix A). Furthermore, I provide and explain my choice of all parameters of the tests and queues, and also detail the considerations that were made during the implementation of the queues. Due to the full source code of

[1] Examples of externally hosted source code becoming unavailable include the original distributions of the source code of the EQueue and B-Queue papers (See footnote 5).

[2] These six queues were selected due to being compared by the EQueue and FastForward papers, with EQueue initially being selected due to having been considered for use in a parallelised network application.

all of these queues being released, all with an identical interface, it will also be easier for implementors to test different queues in their applications.

In the rest of this thesis, I first outline the architecture of the benchmarking framework that was developed and certain technical details of it. Afterwards, I describe the considerations made in the queue implementations, followed by an explanation of the system under test and the different benchmarks that were performed. Then, I analyse the results of the benchmarks, and finally, I present my conclusion.

2 Framework

In this chapter, I will first outline the goals for the framework and will then explain the architectural and technical decisions I made to achieve them.

2.1 Goals

First and foremost, this framework should allow for benchmarking of different SPSC FIFO queues in different benchmarks. In order to make it as straightforward as possible to add new queues or different benchmarks, large code duplications between different benchmarks should also be avoided. For this reason, the framework should be kept modular to allow for simple reuse of existing code. However, duplicating the code shared between benchmarks would potentially allow the compiler to optimise more freely than naïve modularisation would. The details of how this was avoided while still keeping the framework modular are outlined in section 2.3. Lastly, I want to specifically avoid mixing the code being benchmarked and the code responsible for said benchmarking, in order to keep the queue implementations self-contained and easily usable by others.

2.2 Architecture

Typically, benchmarks of FIFO queues are quite similar[3], [4], [8]: Two separate threads will be started, one for enqueuing and one for dequeuing elements (hereafter referred to as the producer and consumer threads, respectively). Both threads will perform an equal number of successful operations, take measurements of certain aspects of the behaviour of the operations, and simulate a workload in between queue operations. What varies between these benchmarks is the type of workload that is simulated and how measurements are taken.

With this in mind, I chose to centre the framework around a `Runner` class, which can be regarded as a "skeleton benchmark." This class implements the aforementioned shared aspects of different benchmarks and makes use of swappable components for the varying waiting behaviour and measurement code. All components are implemented as classes with an interface specific to the type of component they implement.

The producer and consumer threads are quite similar in structure, with the major differences only being related to the queue operation they perform. Both run through a configurable number of iterations, in each of which they repeatedly attempt their queue operation until it succeeds[3] Before and after each operation, the measurement component will be called in order to take measurements, with the measurement from before an operation

[3]This will not cause a deadlock to occur, since a correctly implemented queue will only fail on enqueue / dequeue operations, if the queue is full/empty, respectively.

possibly being repeated in case the operation failed and failed operations are not to be included in the measurements (whether they are to be included is configurable). After each successful operation's post operation measurements, the workload simulation component (hereafter referred to as the waiter or the waiter component) will be called in order to simulate work related to the last performed operation. The waiter will also be called once before the first iteration, so it can adjust wait times based on the start time of the benchmark, in case this is needed for its specific implementation.

As I set out in the previous section, this architecture allows us to implement different benchmarks for different queues, without unnecessary code duplication, by implementing and making use of different components.

2.3 Technical Details

As can be seen in Listing 1, the integration of the components into the runner class was implemented via template parameters.[4] Furthermore, the components are stored as part of the data of the runner class, which, in combination with the components being defined in Header files, means that the compiler will inline both data and method implementations into the runner class. This was verified using the software reverse engineering framework Ghidra. Due to this inlining, the result of compilation will be nearly identical to duplicating the runner class and copying the component's code into it by hand. This means that, as I set out to achieve, the modularisation makes it easier to work with, but it does not impact the performance of the benchmark and should thus not have an impact on the results. Furthermore, the type of the queue implementation is also specified as a template parameter, with the implementation being passed as a parameter to the constructor. Thus, there is also full separation between the queue implementation and the benchmarking code, as the runner class can thus only rely on the common interface of all queues (see Listing 2).

Listing 1 also shows which methods of the waiter and measurer components are being used. First of all, there is the `start` method of the waiter, which marks the beginning of the benchmark, and the `wait` method of the waiter, which triggers a wait time according to the waiter's implementation. Secondly, there is the `pre` method of the measurer, which gets called to record a timestamp before each attempted operation and overwrites the last recorded measurement if called again before the `post` method is called, which is used to take a measurement after each successful operation.

Lastly, it is worth mentioning the main function, which prepares, runs, and cleans up the benchmark. It first loads parameters related to the instantiation of the queues and regarding the benchmark from a config file, and then starts an instance of the runner class with the appropriate components, all according to the CLI arguments it was provided. After the benchmark has finished, it causes the measurement data to be saved.

[4] You may observe the template parameters being pairs of components, this was done to simplify the construction of the runner class and is not relevant here.

```
/* ... */
template <IsQueueWith<u64> Q, IsMeasurerPair M,
    IsWaiterPair W, bool MEASURE_FAILED = false>
class Runner {
  public:
    template <typename P>
    explicit Runner(Q* queue, const P& params)
        : _queue(queue), _measurer(M(params.msg_count)),
          _waiter(W(params)), _msg_count(params.msg_count) {}

    void run() { /* ... */ }

    void write_results(std::ostream* out) { /* ... */ }

  private:
    Q* _queue;
    M _measurer;
    W _waiter;
    usize _msg_count;
    volatile bool _start = false;
    CACHE_ALIGNED u64 _data_sink;

    void producer() {
        // wait for start
        while (!_start) { }
        // do test
        _waiter.tx.start();
        for (usize i = 0; i < _msg_count; i++) {
            _measurer.tx.pre();
            auto data = get_timestamp();
            while (!_queue->enqueue(data)) {
                if constexpr (!MEASURE_FAILED) _measurer.tx.pre();
            }
            _measurer.tx.post();
            _waiter.tx.wait();
        }
    }

    void consumer() {
        // wait for start
        while (!_start) { }
        // do test
        _waiter.rx.start();
        for (usize i = 0; i < _msg_count; i++) {
            _measurer.rx.pre();
            std::optional<u64> data = std::nullopt;
            while (!((data = _queue->dequeue()))) {
                if constexpr (!MEASURE_FAILED) _measurer.rx.pre();
            }
            _data_sink += *data;
            _measurer.rx.post();
            _waiter.rx.wait();
        }
    }
};
```

Listing 1: The implementation of the runner class in C++, with some parts omitted.

2.3.1 Time Measurement

Both the waiter implementations and the measurer implementation need to measure time intervals. The basis of this is the `RDTSC` instruction, which reads the Timestamp Counter (TSC), which is a feature of x86 CPUs. On all CPUs with an invariant TSC, like the one used for the benchmarks here, this counter is incremented at a constant rate, equal to the base frequency of the CPU. This means that in order to provide time interval measurements in nanoseconds, the TSC difference needs to be divided by the CPU's base frequency to arrive at the correct unit.

2.3.2 Waiter Implementations

As outlined previously, waiter components cause a delay between queue operations in order to simulate the enqueue and dequeue patterns caused by real-world workloads. For the benchmarks in this thesis, I have implemented three different waiter components.

Constant Wait

This component waits a constant amount of time between operations. It aims to simulate workloads with very consistent time expenditure per item enqueued or dequeued.

Constant Rate

This component tries to maintain a constant time difference between the starts of queue operations. It is used in order to isolate the differing operation times of different queue implementations from the rate at which operations are performed. For example, if the queue operation takes 80 ns on average, and the wait time (time difference between operations if the specified en-/dequeue rate is maintained exactly) is 200 ns, then Constant Wait will lead to a 280 ns delay between the starts of operations, while Constant Rate will lead to a 200 ns delay between starts of operations, because it would only waits $200\,\text{ns} - 80\,\text{ns} = 120\,\text{ns}$.

Bursty

This waiter behaves similarly to the Constant Rate component; however, it defers wait operations in order to produce blocks of BURST_SIZE operations with little to no delay, similar to network applications where network packets often arrive in bursts. For example, with a BURST_SIZE of 20, a wait time of 100 ns, and an average operation duration of 20 ns, 20 elements will be enqueued, followed by a wait of $20 \cdot (100\,\text{ns} - 20\,\text{ns}) = 1{,}600\,\text{ns}$ (leading to an average delay between starts of operations of 100 ns). Lastly, this component and the Constant Rate waiter also allow for jitter to be enabled, which adds pseudo-random variance to the wait times (this was also done in the FastForward paper[5]).

2.3.3 Measurer Implementations

I have only implemented a single measurer component; however, measurer components could be used to, for example, use different timestamp sources than the `RDTSC` instruction used here, as it is only supported on x86. It could also be used to measure other aspects of

the queue/benchmark behaviour, such as cache statistics, the number of failed operations, etc. Due to being the only implemented measurer, it is used for every benchmark here.

3 Queue Implementations

All of the queues I will benchmark in this work needed to either be ported to C++, adapted to share the same interface (see Listing 2), or reimplemented because no source code was available. In the first section, I will explain how I reached out to authors of those queues for feedback on my implementations. In the following sections, I will outline the considerations that were made for each queue.

3.1 Feedback

On 2025-09-04, I reached out via email to the authors of the MCRingBuffer and FastForward papers, seeking source code for the original implementations or feedback on my implementations of their queues. One of the authors of the FastForward paper responded and was able to provide me with a partial implementation of the FastForward queue, which was missing the slip adjustment code. The provided code was very similar to my implementation, suggesting that part of my implementation is correct. I have not received an answer from the authors of the MCRingBuffer paper as of 2025-09-30; however, my original message could not be delivered to Tian Bu and Girish Chandranmenon since their email addresses (as stated in the MCRingBuffer paper) are not valid anymore.

I reached out via email to Yangfeng Tian and Xiong Fu, who are authors of the EQueue paper, and Junchang Wang, who is an author of the B-Queue and EQueue papers, on 2025-05-28 with some questions about the EQueue paper and source code[5] I have not received an answer to this email as of 2025-09-30, and for this reason, I have not reached out separately about feedback on my implementation.

3.2 BQueue

This queue was initially reimplemented, because the original source code repository linked in the B-Queue paper had become unavailable (see queues/b_queue.h)[3]. As such, a full B-Queue implementation, including self-adaptive backtracking and producer and consumer batching, was made. I have since found a copy of the original repository and validated that my implementation shows the same performance characteristics in my benchmarks. The implementation has two small divergences from the pseudo code in the paper, which are present to fix bugs. First, a check was added to prevent the tail from overtaking the

[5]About a week after reaching out, I noticed that the EQueue repository on GitHub ceased to be available. For preservation purposes, I have reuploaded the repository to GitHub (See Computerdores/equeue). For the same reason, I have also forked a GitHub reupload of the original B-Queue repository, which had already been unavailable prior to my work here (See Computerdores/b-queue).

```cpp
template <typename T>
class Queue {
  public:
    using Item = T;

    // non-blocking on failure
    virtual bool enqueue(T item) = 0;

    // non-blocking on failure
    virtual std::optional<T> dequeue() = 0;

    virtual ~Queue() = default;
};
```

Listing 2: The `Queue<T>` interface.

batch tail. Second, another check was added to prevent the head from overtaking the tail in certain circumstances.[6]

3.3 EQueue

EQueue was ported to C++ with small differences, because the released source code and the paper differed (see queues/equeue.h)[4]. Furthermore, their queue implementation is also not entirely self-contained (see traffic counters). I tried to reconcile the differences between the paper and their code appropriately.

First, I added a check to make sure the used portion of the buffer does not exceed the buffer size, which is also present in their code. Secondly, the structure of `_enqueue_detect_batching_size` differs between the EQueue paper and their implementation; my version of it is closer to their implementation than to the paper. Thirdly, according to the paper, the `traffic_empty` counter should be increased in `dequeue`; however, their implementation does this outside of the queue implementation. To keep my implementation self-contained, I moved this into the `dequeue` method, as specified by the paper. Fourthly, the EQueue paper's pseudo code does not increment `traffic_full` in the batched version of `enqueue`. This seems illogical, since the core mechanism to resize the used portion of the buffer relies on this being incremented. Furthermore, their implementation also increments it, but does so outside of the queue implementation. For my implementation, I increment it as part of the `enqueue` method, similarly to the `traffic_empty` counter. Lastly, I did not implement their LT-CAS primitive, since their implementation does not make use of it as part of the queue either (it only contains a demonstration of the primitive).

[6] This check does not appear to be necessary for the benchmarks anymore; however, I did not remove it because it makes the implementation more robust, and I did not observe any performance difference due to it.

3.4 MCRingBuffer

The queue from the MCRingBuffer paper was implemented from scratch, as there does not seem to be any publicly available source code (see queues/mc_ring_buffer.h)[8]. I made sure to add padding between shared, consumer local, producer local, and read only variables, to prevent false sharing by making sure these blocks of variables are on different cache lines. I also added volatile markings for the variables `read`, `write`, and `buffer`. The last change I made is that when enqueue or dequeue don't immediately succeed, instead of waiting as specified in the paper, the operation instead fails, to conform with the interface[8]. It is also worth mentioning that this queue deadlocks under certain conditions; however, fixing this issue was not pursued here, for the same reasons also outlined in the B-Queue paper[3].

3.5 FastFlow Queue

FastFlow Queue was largely taken as is and only extended to conform with the general queue interface (see queues/ff_queue.h)[6]. This was done since FastFlow Queue has a prepare + commit interface for enqueue and dequeue. To fix this interface discrepancy, wrapper methods were implemented, which make use of the prepare + commit interface to implement the typical enqueue and dequeue semantics.

3.6 FastForward Queue

Like MCRingBuffer, FastForward queue was implemented from scratch, because I was not able to find source code for it (see queues/fast_forward.h)[5]. Also like MCRingBuffer, padding was added to make sure that the `head`, `tail`, and `buffer` variables are on different cache lines, to prevent false sharing. Furthermore, volatile markings and volatile references were added to prevent optimisation errors by the compiler. Lastly, the implementation of the distance function was not specified in the paper and was implemented via access to the otherwise thread-local `head` and `tail` variables.

3.7 Lamport Queue

This queue was also reimplemented due to no source code being available, which had seemingly also been done for the FastForward paper (see queues/lamport.h)[7]. Similar to the other queues, padding was added between the `head`, `tail`, and the read-only variables and volatile references were added as well. It is also worth noting that, as with MCRingBuffer, instead of waiting when an operation does not succeed immediately, like the paper specifies, the queue operations fail instead[7].

4 Benchmarking

In this chapter, I will first give details about the system on which the benchmarks were performed. I will then go on to explain the benchmarks that were performed, and will finally analyse the results of the benchmarks.

4.1 System Under Test

4.1.1 Hardware

The benchmarks were run on a SuperMicro SYS-521C-NR server with an Intel Xeon Gold 6548N CPU with 32 cores running at the base frequency of 2.8 GHz. Each core has 80 kB of L1 Cache, 2 MB of L2 Cache, and 60 MB of L3 Cache shared among the cores. The server has 128 GB of DDR5 memory, in an 8x16 GB configuration running at 4,800 MT/s. All benchmarks were run with the CPU running at base frequency (set via the `cpupower` utility).

4.1.2 Software

The server was running Ubuntu 24.04.2 LTS with Linux Kernel version 6.8.0. The benchmarks were compiled using GCC 14.2.0. Kernel parameters were used to isolate the CPU cores used for the benchmark from influences from the rest of the system. Specifically, the `isolcpus` and `nohz_full` parameters were used.

4.2 Benchmarks

In this section, I will start by explaining the different parameters to be set for each queue. While I will outline the purpose of each parameter, I will not provide an in-depth explanation of the inner workings of each queue. For detailed explanations, please refer to the original paper proposing the respective queues[7] I will then continue to outline the specifics of each benchmark, with one benchmark per subsection in the order from the benchmark with the smallest difference between the performance of the queues to the benchmark with the highest difference between the queues.

[7] Note, however, that the FastFlow paper does not give a detailed explanation of the queue, and only states that its "core is based on efficient Single-Producer-Single-Consumer (SPSC) and Multiple-Producer-Multiple-Consumer (MPMC) FIFO queues, which are implemented in a lock-free and wait-free fashion"[10].

4.2.1 Parameters

BQueue

BQueue implements producer and consumer batching to avoid having to read ahead in the buffer on every operation in order to check if the operation will succeed [3]. BQueue also dynamically adjusts the dequeue batch size. The queue is configured via the following values:

- `size` – The size of the queue.
- `batch_size` – This parameter determines the size of the producer and consumer batches.
- `batch_increment` – This value determines how much larger the candidate for the next dequeue batch size should be than the last used one.
- `wait_time` – While determining the new dequeue batch size, B-Queue waits a certain amount of time when a new candidate is chosen. This parameter determines the duration of each of those waits.

EQueue

EQueue dynamically adjusts the size of the queue to optimise cache access times[4]. Furthermore, EQueue also dynamically adjusts the dequeue batch size. The following parameters influence these behaviours:

- `min_size`, `max_size` – These values determine the valid range for the queue size. The underlying ring buffer will be allocated with a size of `max_size`.
- `initial_size` – The initial size of the queue before the dynamic queue size adjustments.
- `wait_time` – When determining the new enqueue batch size, equeue waits a certain amount of time when a new candidate is chosen. This parameter determines the length of those waits.

FastFlow Queue

FastFlow Queue uses a linked list of dynamically allocated buffers as the storage of the queue[6]. The buffers are referred to as "buckets." This behaviour is controlled by the following parameters:

- `bucket_size` – This parameter sets the size of these buffers.
- `max_bucket_count` – This parameter determines the maximum number of buckets that can be allocated by the queue.

MCRingBuffer

MCRingBuffer implements producer and consumer batching[8]. The queue is configured using the following values:

- `size` – The size of the queue.
- `batch_size` – The size of the producer and consumer batches.

FastForward Queue

FastForward Queue delays dequeues via waiting to try to keep over one cacheline of data in the queue[5]. The distance between head and tail that this creates is called the "slip". The following parameters can be used to control FastForward Queue's behaviour:

- `size` – The size of the queue.
- `adjust_slip_interval` – This parameter specifies the interval, in terms of the number of dequeues, at which the slip is adjusted.

Lamport Queue

The only parameter of Lamport Queue is the `size` parameter, which controls its size[7].

4.2.2 Bursty 65k-16k / Bursty 65k-2k

The Bursty 65k-16k and Bursty 65k-2k benchmarks are based on the tests performed in the EQueue paper, specifically the tests with a burst size of 16,384 and 2,048 elements, respectively. The benchmark performs 1,000,000 operations at a targeted rate for enqueue and dequeue of 1,000,000 operations per second.[8] All benchmarks (including Basic 65k) were repeated 100 times, with the operation times being averaged for every repetition.

The benchmark uses the Bursty waiter component to simulate bursty enqueue patterns, similar to the EQueue paper. The burst sizes of 16,384 and 2,048 were selected because both were tested in the EQueue paper and showed very different results. The concrete selection of the burst size 16,384 over the burst size 32,768 was arbitrary due to the small difference between the reported results. It is further worth noting that the implementation of the bursty enqueue pattern differs slightly from the one used in the EQueue paper, because the Bursty waiter adjusts for the time the operation takes, as outlined in subsection 2.3.2. The waiter for the consumer thread is a Constant Wait component, selected because it is closest to the implementation from the EQueue paper.

The minimum and maximum sizes of EQueue are set to 256 and 65,536, respectively, as also stated in the EQueue paper. The paper does not specify the queue sizes used for queues other than EQueue. For this reason, I chose 65,536 as the (maximum) queue size for all other queues in order to have a like-for-like comparison. In the next paragraphs, I will outline the reasons for the remaining queue parameter values. The configuration containing all of these values can be seen in Listing 3. The full configurations for both of these benchmarks can be found in `equeue_repro_16k.toml` and `equeue_repro_2k.toml`, respectively.

BQueue

The `batch_size` parameter was set to a sixteenth of the queue size as specified by the original B-Queue source code. In turn, the `batch_increment` parameter was set to half

[8] This rate differs from the rate used in the EQueue paper, where they targeted $\approx 20,000,000$ operations per second (specified as a wait time of 85 cycles $\approx$ 50 ns). The differing rate was selected because, according to the results from the EQueue paper, some, but not all, of the queues tested would not be able to operate at a rate of 20,000,000 operations per second, due to their operation time being larger than 50 ns[4].

```
[BQueue]
size = 65536
batch_size = 4096
batch_increment = 2048
wait_time = 358

[EQueue]
initial_size = 8192
min_size = 256
max_size = 65536
wait_time = 358

[FastFlow]
bucket_size = 4096
max_bucket_count = 16

[MCRingBuffer]
size = 65536
batch_size = 8000

[FastForward]
size = 65536
adjust_slip_interval = 64

[Lamport]
size = 65536
```

Listing 3: The Queue parameters for the Bursty 65k benchmarks.

of the `batch_size` parameter, as also specified by the source code. Lastly, the `wait_time` parameter was set to 358 ns in accordance with the 1,000 cycles value from the source code.

EQueue

For the `initial_size` value of EQueue, no value was specified in the paper, and it was thus tuned from scratch[9] The other unaccounted for parameter, `wait_time`, was specified in the paper and source code as 1000 Cycles and was thus set to the equivalent 358 ns.

FastFlow Queue

For FastFlow queue, there are two parameters: `bucket_size`, for which there is no specified default value and which was thus tuned from scratch, and `max_bucket_count`, which was entirely predetermined by the bucket size and the targeted maximum size of the queue.

MCRingBuffer

The `batch_size` parameter was tuned – with the constraint that it needed to be a divisor of the number of enqueue / dequeue operations in order to avoid the aforementioned deadlock issue.

FastForward Queue

The value for `adjust_slip_interval` was taken from the FastForward paper as is and then verified via tuning.

Lamport Queue

The Lamport queue has no parameters, except for the `size` parameter.

4.2.3 Basic 65k

The Basic 65k benchmark is identical to the Bursty benchmarks, except that the Constant Rate waiter was used for the enqueue thread instead of the Bursty waiter. The EQueue paper does not include a test with a constant rate or constant wait enqueueing, but this benchmark is more similar to the testing in B-Queue and FastForward queue than the Bursty benchmarks are. The configuration file for this benchmark is `equeue_repro_16k.toml`.

4.3 Results

It can generally be noted that the jitter on the TX side does not have an impact on any of the results (See figs. 4.1 to 4.3). Jitter having a significant impact can be a problem where the precise timing of the benchmark causes significant changes in performance. This was also observed at times during the development of the benchmarks, but disappeared after applying bug fixes.

[9] Note that all of the parameter tuning was done at significantly lower sample counts than the final benchmarks, because of the unfeasibly large amount of time this would otherwise have taken.

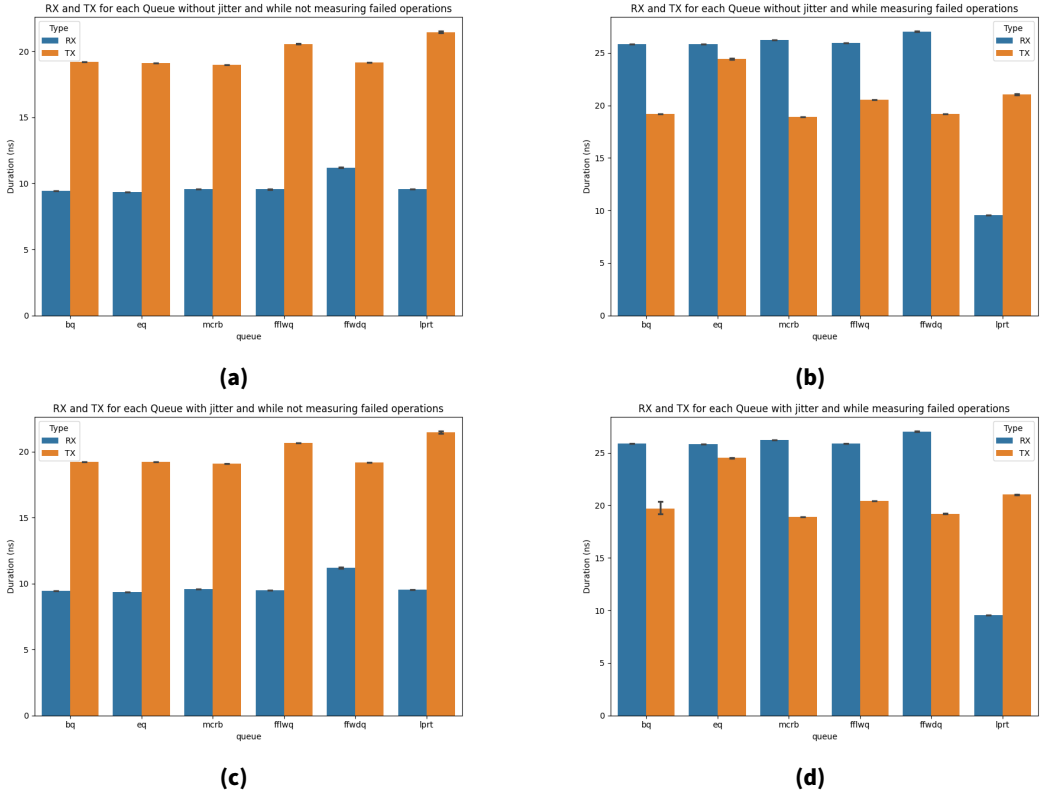

Figure 4.1: Results for the Bursty 65k-16k benchmark, with error bars showing the 95% confidence interval.

Furthermore, regarding comparisons to the EQueue paper, three things are of note. First, the measurements from the EQueue paper include parts of their measurement code; my measurements do not[11]. However, it is not clear to me how significant the impact on their measurements would be. Second, I suspect that the measurements with different burst sizes from the EQueue paper were done in their cross-cpu configuration, which would be a significant difference from my measurements, but the paper does not specify whether or not this is the case. I suspect this because the values for burst size 2,048 from the graph match the cross-cpu values from their table better than the same-cpu values. Third, the values for the different burst sizes are not directly stated and were therefore discerned from the graph included in the paper.

In the following subsections, I analyse the results from the different benchmarks. I first address the two bursty benchmarks and compare the results to the results from the EQueue paper, as they also used bursty enqueue timing. Lastly, for the Basic benchmark, I compare my results to those of the B-Queue and FastForward papers, as this benchmark is, out of mine, most similar to those.

4.3.1 Bursty 65k-16k

As mentioned in chapter 2, the components give two options to be varied with each benchmark: jitter and whether failed operations are included in the measurements. The four variations of this benchmark can be seen in Figure 4.1. The variation in Figure 4.1b, without jitter and while including failed operations in the measurements, is closest to the burst size 16,384 test from the EQueue paper[4].

According to the EQueue paper, EQueue reportedly took $\approx$ 60 cycles per operation, which is $\approx$ 36 ns. My measurements show a time per operation for EQueue of $\approx$ 24.4 ns for enqueue and $\approx$ 25.8 ns for dequeue. These results suggest that my EQueue implementation does not exhibit major performance flaws, because the observed difference can be adequately explained by the known differences between these test setups. It also gives credence to the idea that the impact of the differences between the benchmark and the EQueue paper's testing is limited.

The other queues tested in the EQueue paper (B-Queue, MCRingBuffer, and FastForward Queue) reportedly took around 112 ns per operation. In my measurements, these queues all performed similarly to EQueue at around 19 ns–21 ns per enqueue operation and around 26 ns–27 ns per dequeue operation. This represents a significant difference between my measurements and those reported in the EQueue paper. The reasons for this difference are unclear. It could, for example, be explained by differences in parameters for the other queues or by hardware differences that specifically favour EQueue on the hardware used in that paper. Another possible explanation is the difference in the rate at which operations are performed; however, further research would be required to determine the exact causes with confidence.

It can also be observed that all queues, except for Lamport queue, show a $\approx$ 50% better dequeue operation time when not including failed operations in the measurements. This means that a significant portion of the dequeue operations of these queues are failing in this benchmark. I suspect that this happens during the wait time between bursts and thus would not lead to decreased application performance in a real-world scenario.

Overall, it can be observed that all queues perform similarly in this benchmark, except for Lamport queue, which performs better than the other queues on the dequeue operations. However, it is not clear whether this would lead to a real-world performance advantage for Lamport queue.

4.3.2 Bursty 65k-2k

The results from the four variations of this benchmark can be seen in Figure 4.2. As with the previous benchmark, the variant of this benchmark without jitter and while including failed operations in the measurements is the closest to the test from the EQueue paper with burst size 2,048 (see Figure 4.2b).

According to the EQueue paper, at a burst size of 2,048, all queues tested there (EQueue, B-Queue, MCRingBuffer, and FastForward Queue) took about 48 cycles, which is about 29 ns. In my measurements, all queues except for EQueue perform similarly to each other at around 10 ns–21 ns. This difference to the EQueue paper is about what would be expected, since it lines up with the one observed for EQueue in the previous benchmark and can therefore also be explained by the known differences between the benchmarks. However,

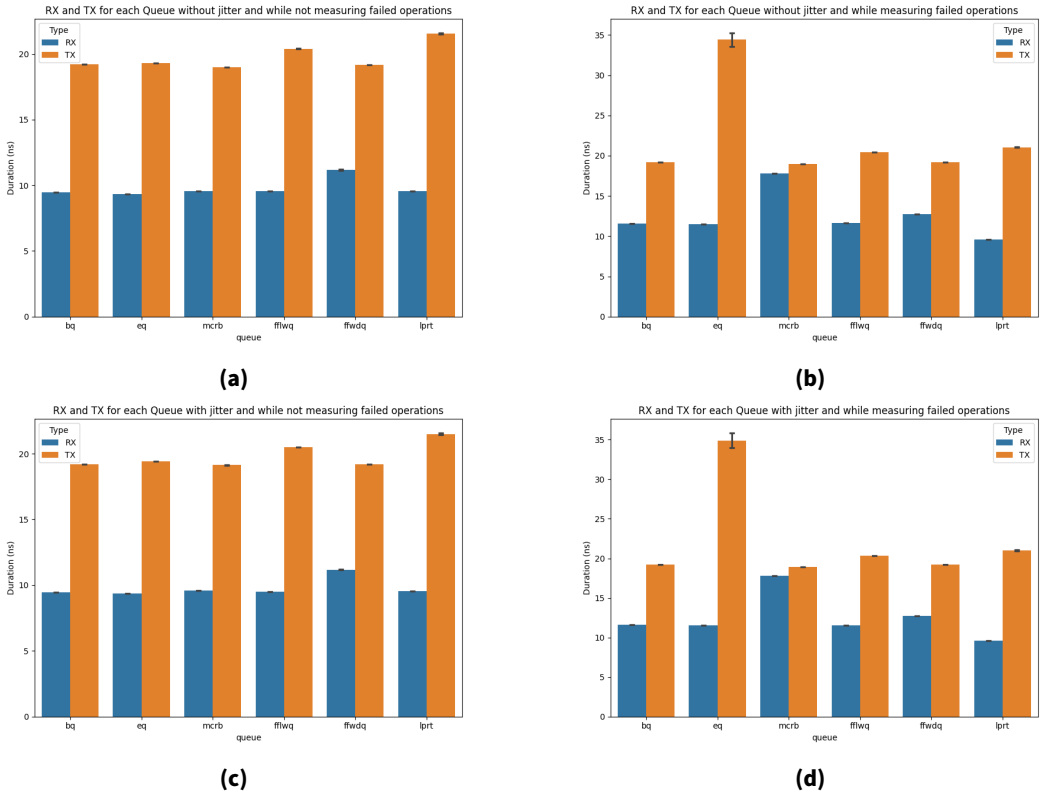

Figure 4.2: Results for the Bursty 65k-2k benchmark, with error bars showing the 95% confidence interval.

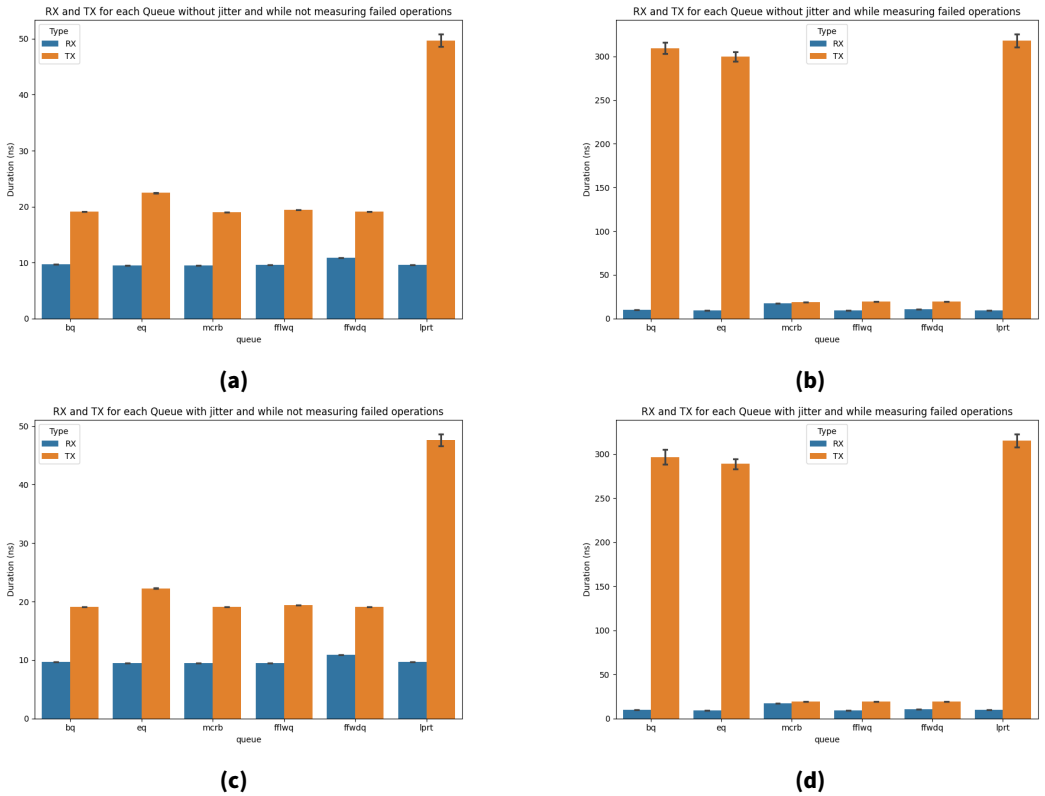

Figure 4.3: Results for the Basic 65k benchmark, with error bars showing the 95% confidence interval.

at $\approx$ 34.4 ns per enqueue operation, EQueue performs significantly worse than the other queues in my measurements. When not measuring failed operations, this difference can not be seen, which means that it appears due to failed enqueue operations. On its own, this could be explained by there being fewer than 2,048 free slots in the queue, leading to some of the last elements from each burst not immediately fitting into the queue, thus leading to failed enqueue operations. However, this effect would also be expected to be more pronounced with a higher burst size, which can not be observed in the previous benchmark. It is thus not clear why there is a higher number of failed operations for EQueue here than there is for the other queues. Lastly, MCRingBuffer also seems to experience an increased number of failed dequeue operations over the other queues. It is again not clear why this is the case, as successful queue operations of MCRingBuffer are similarly fast to the other queues.

Overall, it can be said that the queues all perform relatively similarly in this benchmark, with the enqueue operation of EQueue as an outlier and the dequeue operation of MCRingBuffer as another, smaller outlier.

4.3.3 Basic 65k

For this benchmark, the results of the four variations can be seen in Figure 4.3. In the case of the results of the variation without jitter and while measuring failed operations, it can be

observed that most of the results are almost identical to those of the Bursty 65k-16k benchmark, with the enqueue operation time of B-Queue, EQueue, and Lamport Queue being outliers, which are approximately one order of magnitude worse than in the Bursty 65k-16k benchmark. Comparing those results to the variation with neither jitter nor measuring of failed operations, this increase in operation time can only be seen for Lamport queue and only to a smaller degree. This means that while EQueue and B-Queue are only outliers due to failed operations, for Lamport queue, the duration of successful enqueue operations is actually higher than in the other benchmarks. This could mean that some difference in timing leads to EQueue, B-Queue, and Lamport queue filling up early in the benchmark, leading to failed enqueue operations later on because the queues are full. As for the increase in duration for Lamport queue's enqueue operations, which isn't present for the other queues, this could be due to cache optimisations, as Lamport queue is the first concurrent lock-free queue to be presented, with the other queues all focusing on optimisations over existing queues, which Lamport queue does not have[7].

This benchmark is also, to a degree, similar to the testing done in the FastForward paper, where FastForward queue and Lamport queue were compared. However, in the FastForward paper, three cores were set up to forward in a circle, with each pair of cores being connected by a queue, which presents a significant difference from this benchmark[5]. Furthermore, in this benchmark, the queues are of much larger size, which is another difference from the FastForward paper's testing. Despite those differences, my measurements confirm that Lamport queue does indeed perform much worse than FastForward queue, both when measuring failed operations and when not doing so, as was found in the FastForward paper. However, the performance is not identical to that measured in the FastForward paper, as could reasonably be expected given the differences in testing setups.

This benchmark is also the most similar one, out of the ones included, to the testing in the B-Queue paper with one concurrent queue. However, the testing in the B-Queue paper used much smaller queues, so some differences could be expected due to that. The B-Queue testing showed operation times of around 8 ns for B-Queue, MCRingBuffer, and FastForward queue. My measurements also show that the operation times for FastForward queue, MCRingBuffer, and the dequeue operation of B-Queue are very similar to each other; however, the absolute operation times are at around 10 ns–11 ns for dequeue operations and at around 19 ns for enqueue operations. Since the B-Queue measurements were obtained by measuring the time, performing all queue operations, and measuring the time again, for them, the maximum operation time between the enqueue operation time and the dequeue operation time is what they measured as the general operation time. Applying this to my measurements leads to an effective general operation time of $\approx$ 19 ns, in contrast to the $\approx$ 8 ns from the B-Queue paper, meaning the B-Queue measurements are lower than any result from my measurements or the measurements from the EQueue paper for any of the three queues[3], [4]. This suggests that the absolute values measured in the B-Queue paper are not comparable to the results from my measurements or those from the EQueue paper. It is not clear what causes this difference; however, it could be due to hardware differences, as both the EQueue paper and I used significantly newer CPUs for the testing than the B-Queue paper did.

Overall, these results suggest that in scenarios with low fluctuation of the enqueue rate, the performance of FastForward queue, FastFlow queue and MCRingBuffer is best out of the queues tested here.

5 Conclusion

This paper presents a framework for testing the performance of SPSC FIFO queues. This framework simplifies the process of testing the performance of those queues by only requiring small modifications to the main function of the framework, and that the queue implements a common interface. Through its modular architecture, the framework also makes it easier to test the queues under multiple different conditions. Furthermore, parameters for queues and benchmarks are stored in config files, making sets of them simpler to manage and to keep different versions of. This framework was used in this thesis to test different queues and has shown its ease of use in the process. Collectively, this framework should lessen the work required of future authors to test proposed queue designs in a replicable and comparable manner.

In this work, I also tested the performance of six different queues in three different benchmarks with four variations each. In part, the results from these tests show significant similarities to those presented in the EQueue, B-Queue, and FastForward papers; however, there are also significant differences that were observed[3], [4], [5]. Several potential reasons for these differences were identified. First, there is the possibility that the results from the EQueue paper were measured in their cross-CPU configuration, which would present a significant difference from the benchmarks performed here. Second, the parameters of the other queues benchmarked in the EQueue paper are not known and might thus be significantly different to those used here. Third, the three papers did not specify what steps they took to isolate outside influences on their results, and it is thus not clear how those differ from the approach taken here. Fourth, inevitably, the results of all papers were obtained on different hardware, which, especially for the B-Queue paper, might also be a reason for the differences. In writing this paper, I have attempted to minimise the issues mentioned above, and those not repeated here, by publishing the full source code required to replicate my measurements, providing detailed contextual information, and outlining my reasoning for key decisions made in the process.

Given that FastForward queue and FastFlow queue were among the queues with the best performance in almost every benchmark, it is recommended that they be the first candidates when choosing a queue implementation. Similarly, MCRingBuffer performed only slightly worse than these two queues and, therefore, also makes for a good candidate. However, given the differences between the results found here and in other papers, it is still recommended to test multiple queues in the concrete application in which the chosen queue will be used. This should also require a reduced amount of work, due to the queue implementations tested here all implementing the same interface and having their source code published. Lastly, should the number of queues that can be tested be limited, the results presented here should give application developers insight into which candidates to prioritise.

Building on the work done in this thesis, future contributions could include additional benchmarks featuring, for example, multiple producer and consumer threads on the same

core or more realistic simulated workloads. Additionally, more sophisticated analysis of the data produced by the benchmarks could also be done. This could include analysing the difference between completed enqueue and dequeue operations, or the number of failed operations, over time. Lastly, a measurer component could be implemented to measure cache statistics, specifically the cache miss percentage, which was not implemented in this thesis due to time constraints. This was done in other papers, including the EQueue and B-Queue papers, and could be interesting to investigate the performance differences between the queues. However, these cache statistics should be measured per operation and not include the benchmarking code, in line with the existing measurements. This might be possible using the oprofile tool, which was also used in the B-Queue paper; however, due to function inlining, a different approach might be needed.

Bibliography

[1] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of parallelizing network applications on multi-core architectures", in *Proceedings of the 23rd International Conference on Supercomputing*, series ICS '09, Yorktown Heights, NY, USA: Association for Computing Machinery, 2009, pages 204–213. DOI: 10.1145/1542275.1542307.

[2] G. Upadhyaya, V. S. Pai, and S. P. Midkiff, "Expressing and exploiting concurrency in networked applications with aspen", in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, series PPoPP '07, San Jose, California, USA: Association for Computing Machinery, 2007, pages 13–23. DOI: 10.1145/1229428.1229433.

[3] 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*, volume 41, pages 137–159, Aug. 2012. DOI: 10.1007/s10766-012-0213-x.

[4] J. Wang, Y. Tian, and X. Fu, "EQueue: Elastic lock-free fifo queue for core-to-core communication on multi-core processors", *IEEE Access*, volume 8, pages 98 729–98 741, 2020. DOI: 10.1109/ACCESS.2020.2997071.

[5] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue", in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, series PPoPP '08, Salt Lake City, UT, USA: Association for Computing Machinery, 2008, pages 43–52. DOI: 10.1145/1345206.1345215.

[6] FastFlow contributors, *FastFlow pattern-based parallel programming framework (formerly on sourceforge)*, 2017. Accessed: Sep. 28, 2025. [Online]. Available: https://github.com/fastflow/fastflow.

[7] L. Lamport, "Specifying concurrent program modules", *ACM Trans. Program. Lang. Syst.*, volume 5, number 2, pages 190–222, Apr. 1983. DOI: 10.1145/69624.357207.

[8] P. P. C. Lee, T. Bu, and G. Chandranmenon, "A lock-free, cache-efficient shared ring buffer for multi-core architectures", in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, series ANCS '09, Princeton, NJ, USA: Association for Computing Machinery, 2009, pages 78–79. DOI: 10.1145/1882486.1882508.

[9] F. Fidler and J. Wilcox, "Reproducibility of Scientific Results", in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Summer 2021, Metaphysics Research Lab, Stanford University, 2021. Accessed: Sep. 30, 2025. [Online]. Available: https://plato.stanford.edu/archives/sum2021/entries/scientific-reproducibility/.

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating code on multi-cores with FastFlow", in *Euro-Par 2011 Parallel Processing*, E. Jeannot, R. Namyst, and J. Roman, Eds., Springer Berlin Heidelberg, 2011, pages 170–181. DOI: 10.1007/978-3-642-23397-5_17.

[11] J. Wang, Y. Tian, and X. Fu, *EQueue source code repository*, https://github.com/Computerdores/equeue/ (originally at https://github.com/junchangwang/equeue, now unavailable), 2020. Accessed: Sep. 30, 2025.

A Source Code Archive

As mentioned previously, to ensure the preservation of the source code of everything related to this work, the complete repository was embedded into this document as an archive. This also includes analysis reports with the data presented by the figures, among other things.

The archive can be extracted via the following link: src.tar.gz. Since this is not a commonly used feature of PDF files, it may not work in every PDF viewer. It was tested and confirmed to work in Firefox version 141.0 on Linux.

B Search Queries

To find papers that compare different SPSC FIFO queue implementations, I performed the following queries using Google Scholar and examined the first five pages of the results for each query:

- `SPSC FIFO queue benchmark comparison`
- `FIFO queue "benchmark"`
- `FIFO queue comparison`
- `FIFO queue performance evaluation`
- `"multi-core" "queue" benchmark`
- `B-Queue`

These results did not include any papers focusing on comparing different implementations; however, the results did contain several papers presenting novel queue designs, which may be of future interest:

- *A Memory Efficient Lock-Free Circular Queue*, DOI: 10.1109/ISCAS51556.2021.9401239
- *A cache-friendly concurrent lock-free queue for efficient inter-core communication*, DOI: 10.1109/ICCSN.2017.8230170
- *Correct and Efficient Bounded FIFO Queues*, DOI: 10.1109/SBAC-PAD.2013.8
- *The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU*, DOI: 10.1145/3205289.3205291
- *Using elimination to implement scalable and lock-free FIFO queues*, DOI: 10.1145/1073970.1074013
- *An Efficient Unbounded Lock-Free Queue for Multi-core Systems*, DOI: 10.1007/978-3-642-32820-6_65
- *Enabling Extremely Fine-grained Parallelism via Scalable Concurrent Queues on Modern Many-core Architectures*, DOI: 10.1109/MASCOTS53633.2021.9614292
- *Fast concurrent queues for x86 processors*, DOI: 10.1145/2442516.2442527
- *The Baskets Queue*, DOI: 10.1007/978-3-540-77096-1_29
- *A wait-free queue as fast as fetch-and-add*, DOI: 10.1145/2851141.2851168
- *wCQ: A Fast Wait-Free Queue with Bounded Memory Usage*, DOI: 10.1145/3490148.3538572

C Kurzfassung

First In, First Out (FIFO)-Queues spielen eine wichtige Rolle in der Implementierung von parallelisierten Architekturen als Kommunikationskanäle zwischen unterschiedlichen Anwendungskomponenten[1]. Diese Kommunikationskanäle können oft zentral für die Leistung solcher Anwendungen sein, insbesondere für Netzwerkanwendungen, da diese häufig Pipeline-Parallelismus nutzen[1], [2]. Es gibt jedoch viele unterschiedliche Implementierungen von Single Producer Single Consumer (SPSC) FIFO-Queues, die jeweils unterschiedlich getestet werden, was Vergleiche erschwert. Aufgrund dessen ist es schwierig, die optimale Queue für einen Anwendungsfall zu wählen, ohne alle in Betracht gezogenen Kandidaten zu implementieren und zu testen. Um diesem Problem entgegenzuwirken, entwickle ich ein Benchmarking Framework, welches das Testen unterschiedlicher Queues erleichtern und zukünftigen Autoren helfen sollte, ihre Queuedesigns in einer replizierbaren und vergleichbaren Weise zu testen. Außerdem nutze ich dieses Framework, um sechs verschiedene Queue-Implementierungen zu testen: B-Queue, EQueue, FastForward Queue, FastFlow Queue, Lamport Queue und MCRingBuffer [3], [4], [5], [6], [7], [8]. In allen hier genutzten Benchmarks schneiden FastForward Queue, FastFlow Queue und MCRingBuffer (mit) am besten ab. Weiterhin vergleiche ich die Ergebnisse der hier durchgeführten Benchmarks mit denen der Artikel zu EQueue, B-Queue und FastForward Queue, welche ähnlich sind, jedoch auch signifikante Unterschiede aufweisen. Außerdem veröffentliche ich den gesamten hier genutzten Quellcode, zusammen mit zusätzlichen Kontextinformationen, um die Replikation meiner Ergebnisse zu erleichtern.

Glossary

FIFO First In, First Out. ii, 1, 3, 21, 26, 27

SPSC Single Producer Single Consumer. ii, 1, 3, 21, 26, 27

TSC Timestamp Counter. 6

List of Figures

List of Listings