Block-Based Queue – A Novel Approach to Relaxing a FIFO

Bachelor's Thesis of

Stefan Tobias Koch

At the KIT Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

First examiner:	Prof. Dr. Peter Sanders
Second examiner:	T.T.-Prof. Dr. Thomas Bläsius
First advisor:	M.Sc. Marvin Williams

27. May 2024 – 27. October 2024

Acknowledgments

First of all I would like to thank my advisor Marvin Williams for making my transition from sequential to highly concurrent programming much easier with his knowledge and experience in the field. His insightful advice and many fruitful discussions allowed me to mold the BBQ into the respectable data structure that it is today, as well as enabling me to appropriately compare it with its competitors in this thesis. Additionally, I gratefully acknowledge the code he has provided me with, including a data structure for quality analysis, the MultiFIFO as a noteworthy competitor and a parallel implementation of Dijkstra's algorithm along with various graph instances I could use for benchmarking. I would also like to thank Prof. Peter Sanders for providing valuable ideas during the design of the data structure, which I could integrate to great effect. I also thank the entire Algorithm Engineering research group at the Institute for Theoretical Informatics at KIT for allowing me access to their powerful compute machines for benchmarking. Lastly I would like to thank my family and friends for their continued support and a bit of proofreading and running by of some ideas here and there.

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 27.10.2024

. . . .

(Stefan Tobias Koch)

Abstract

We present a novel approach to designing a highly scalable queue based on the relaxation of its FIFO semantics. We use a circular buffer that we partition on two levels to allow for spreading the contention on the singular head and tail of a non-relaxed queue among multiple heads and tails within one section of the relaxed queue while benefiting from cache locality. We also introduce a bitset as an acceleration structure for locating blocks. We then evaluate our queue in different configurations to establish sets of parameters that represent different compromises between quality and performance. Finally we compare our queue against other, primarily relaxed, implementations in multiple micro and one macro benchmark.

Contents

1. Introduction

First-in-first-out is a principle that we encounter in life on an almost daily basis, be it the queue at the supermarket or the concept of first-come-first-served at a restaurant. It seems only logical and fair for requests to be handled in the order that they arrive. Even the concept of relaxing these strict semantics is not without parallel in real life, opening another register at a supermarket based on demand means people who arrived later may be handled earlier than some who arrived earlier, but the throughput of the entire system grows twofold! This is essentially the premise behind utilizing relaxation to improve performance in parallel systems. We have contention on a singular points (one cashier, one end of the queue) and seek to spread it among multiple (multiple cashiers, multiple potential queues to join) in order to decrease the contention on those singular points.

FIFO-queues are one of the most fundamental data structures in computer science, although their implementation is not completely trivial, as they need to support insertion at the back while removal from the front, which does not lend itself well to a linear memory model. In order to implement them, there exist two fundamental approaches, one uses an unbounded-size linked list of nodes, where their order in memory is irrelevant as they directly point to one another; the other uses a bounded-size ring-buffer, where we simply say that element 0 is at the same time our element N, 1 also being $N + 1$ and so on, meaning our queue "runs around" the same chunk of memory over and over again. This separation of approaches similarly applies to the concurrent and relaxed cases.

The uses of queues in practical applications and algorithms is vast, as the fairness of the FIFO semantics is oftentimes a desirable characteristic. Examples include web servers, which often have to handle immense amounts of requests, ideally in a way that prioritizes older requests over newer ones, or breadth-first search (BFS) graph algorithms. Both of these examples are of particular interest: Web servers because they 1. require concurrency for acceptable performance, which may mean a slow concurrent queue can become a bottleneck and 2. don't require strict FIFO semantics since it is usually not a huge concern if requests are handled slightly out of order. BFS algorithms are of interest because they sometimes have the capability of being adapted into relaxed concurrent variations that produce sub-optimal solutions at much higher scalability and performance. The sub-optimal solutions can then either be used directly if that is acceptable for the use case, or they can sometimes be made into optimal solutions with slightly more work.

Implementing a queue that is both efficient and scalable while providing concurrent access proves inherently difficult, as their semantics require coordination in modifying the tail and head of the queue, leading to high contention which may pose a significant bottleneck in parallel applications [2].

A simple approach to alleviating this issue is by relaxing said semantics: popping elements only "somewhere" at the back and pushing elements only "somewhere" at the front allows achieving a speedup by spreading the contention on the singular back and front among multiple locations. This approach is sensible, because the order of operations for independent threads operating on the same shared data structure concurrently cannot be strictly defined [9] and as seen above, strict adherence to the FIFO semantics is often not required by the application context.

The specific approach taken by this thesis intends to use a single contiguous circular buffer of fixed size which is shared among the threads. The primary idea behind our approach is allowing the majority of operations of every one thread to occur within a contiguous block of memory that is ideally not shared with other threads, benefiting from both cache locality and a lack of contention. These blocks can be seen as "mini-queues", as they contain both a head and a tail and within them FIFO semantics are maintained. Additionally, we use a bitset as an acceleration data structure to speed up the finding of blocks that are fit for either pushing or popping.

We focus our design on being very resilient to extreme contention scenarios while expecting a certain amount of elements within the queue at all times for the expected improvements to performance.

2. Preliminaries

A FIFO (first-in first-out) queue of element type `T` is a data structure with two primary operations:

- `bool push(T)`
- `Optional<T> pop()`

Push and pop are also referred to as dequeue and enqueue respectively. A "writer" executes a push operation while a "reader" executes a pop operation.

The semantics of such a queue dictate that the element received from a pop operation is the element that has been in the queue for the longest time, as in, has been pushed at the earliest without having been popped yet.

Additional operations that may be defined are the non-mutating emptiness and fullness checks, which check whether the queue contains any elements to pop or is free to be pushed to respectively and the peek operation, which returns the next element to be popped without mutating the queue. These additional operations tend to have questionable semantics and usefulness within a concurrent context as well as being non-trivial to implement in a linearizable way.

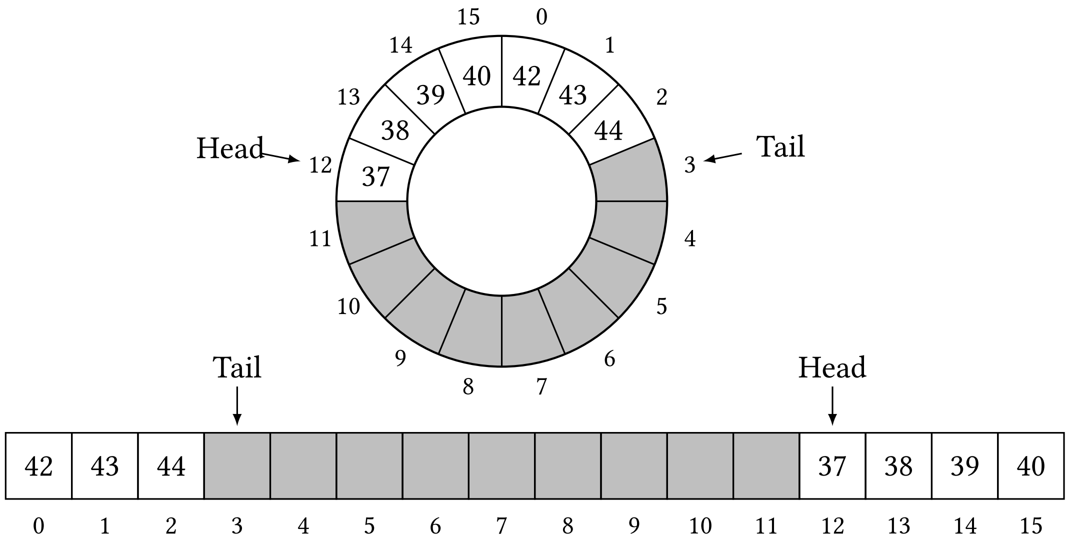
We restrict `null`, 0 and equivalent values from being added to the queue as this provides an invalid state for cells of the queue that may be used in design.

2.1. Ring Buffers

Intended to model an infinite array of which only a contiguous portion of limited size is ever in use at once, ring buffers are commonly used to implement data structures such as queues. They use a fixed-size array as a base on which they operate on using head and tail pointers or counters. For a queue of size S the head and tail must be less than S elements apart. Indexing into the physical array is accomplished via calculating the indices of the modeled infinite array modulo S. The benefit over linked-list based approaches is the locality of memory accesses due to the array's contiguity, which allows benefiting from CPU caching, see 2.3.

See Figure 2.1 for a visualization of the concept.

Figure 2.1.: Ring buffer and its backing array with $S = 16$ filled with 6 elements



2.2. Atomics

Based on the ISO C++ standard [7].

Modifying a regular variable from multiple threads simultaneously is undefined behavior in C++. To still allow concurrent mutation of variables the C++ standard library provides atomic variables, which act as wrappers around regular datatypes and allow well-defined mutation in a concurrent context. Atomic variables offer the best performance when they are implementable via specific CPU instructions and do not require locks. This is usually the case if they are well-aligned and at maximum 64 bits in size. All mutable primitive data members within our implementation are atomic. Besides the expected operations of loading and storing, atomics offer certain operations that only make sense in the face of concurrent mutation and are employed throughout our implementation. Those utilized within our implementation are outlined below.

bool compare_exchange_{strong, weak}(T& expected, T new) Commonly referred to as *compare-and-swap* and abbreviated to CAS, this operation has no equivalent in non-atomic variables. It performs one of two operations, depending on whether the value stored within the variable equals the `expected` value at the time of execution. If it does equal the `expected` value, the variable's value will be replaced by the `new` value, if it does not, the `expected` value will be set to the variable's actual value. It returns true if the variable's value has been updated to the value of `new`, false otherwise. CAS allows performing any desired mutation to an atomic variable, independent of its complexity, by executing it in a loop as seen in Algorithm 2. Beneficiary to this purpose, C++ provides two variants of `compare_exchange`: `strong` and `weak`. `strong` features the behavior one would expect, failing only exactly when the value of the variable does not match the value of `expected`, while `weak` may fail without reason, which it makes up for by being generally more performant on certain architectures. This means when using CAS in a loop the weak variant is generally to be preferred.

Figure 2.2.: Basic atomic algorithms

Algorithm 1: CompareAndSwap

```
Input: T& $expected$, T $new$
if $this = expected$ then
    $this \leftarrow new$;
    return $true$;
else
    $expected \leftarrow this$;
    return $false$;
```

Algorithm 2: MutateAtomically

```
Data: Atomic<T> $v$, Mutator $f : T \rightarrow T$
$expected \leftarrow$ Load($v$);
repeat
    // Noop
until CAS($v$, $expected$, f($expected$));
```

T fetch_{add, sub, or, and}(T other) Equivalent to +=, -=, |= and &= respectively. Applying the mutation atomically on the value of the variable, whichever value that might be. Returns the value of the variable before the mutation, allowing reconstruction of the exact mutation that was undertaken.

2.3. Cache Effects

Based on *The Art of Multiprocessor Programming* [9].

Modern processors utilize hierarchical caches to handle data. Understanding how they work and how to work with them is essential in writing performant concurrent code.

On the most basic level every processor on a multi-processor system needs to load memory into its own non-shared cache in order to work with it, if it needs to access data that is not within its cache then it must load that data from the main memory, which takes *some* time. Data is only loaded into the cache in complete cache lines, meaning no matter how small the amount of data that is required is, all the data surrounding it will be loaded into the cache as well. Thus, by placing data that is often used together in time next to each other in space allows for saving the load time of that data from main memory into the cache. This is referred to as **cache locality** and is a major factor why queue implementation utilizing ring buffers which are in contiguous memory perform better than implementations relying on linked lists which are usually scattered in memory.

However, different data being stored within one cache line can also be detrimental to performance due to **cache coherence**. Cache coherence is a concept that requires multiple processors holding the same cache line to maintain consistency of the data within that cache line. This means every write by one of those processors to the cache line must be communicated to all other processors holding that cache line in some way, taking *some* time. Thus, even if processor 1 only ever uses A and processor 2 only ever uses B, if A and B share a cache line every write to either of them will need to be communicated to the other processor. This disadvantageous cache effect is called **false sharing** and is something that can be avoided by aligning and padding data in a way that would result in A and B being on different cache lines.

We derive two simple, general guidelines:

1. Store data that is used together close together
2. Spread data that is accessed concurrently but not actually shared between processors across different cache lines

2.4. Serializability, Inter-Thread Serializability and Linearizability

Serializability, inter-thread serializability and linearizability are properties of a set of operations in relation to a definition of consistency among them. By extension they can be applied to a concurrent data structure if all possible sets of operations that can be generated by said data structure fulfill the property in relation to the data structure's invariants. Each property is more strict than the previous one, so:

$$\text{Linearizability} \implies \text{Inter-Thread Serializability} \implies \text{Serializability} \tag{2.1}$$

The opposite implications are not necessarily true. We defer full formalization to the relevant literature and only give a brief summary to provide a rough understanding of the concepts. [9]

Serializability. A set of operations is serializable if the operations can be reordered in a manner so that they could occur by a serial execution without harming any of the invariants. [12]

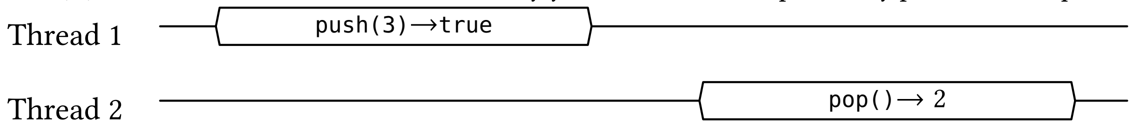
Inter-Thread Serializability. A set of operations is inter-thread serializable if the reordering of operations required for serializability can be achieved without changing the order of operations within a single thread of execution.

Linearizability. A set of operations is linearizable if the only operations that need to be reordered to achieve serializability are those that are actually executed concurrently. Linearization can be imagined as assigning each operation that spans a certain amount of time a single point within that timeframe and having the resulting serial set of operations be consistent. [5]

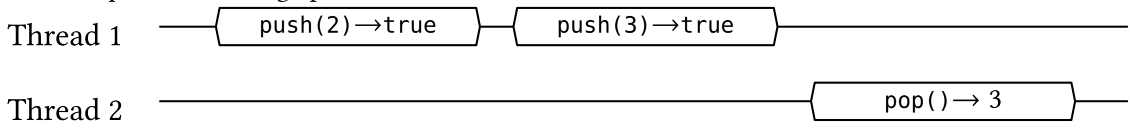
See Figure 2.3 for an illustration of the concepts.

Figure 2.3.: A series of concurrency diagrams intended to illustrate the differences between the different concurrent consistency models by example of operations on an empty queue

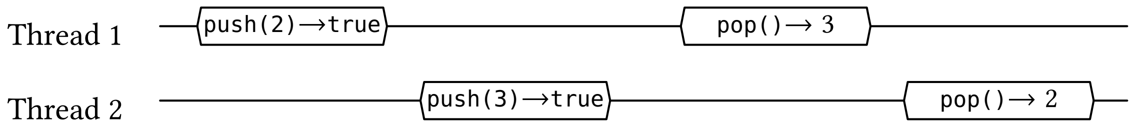
(a) Not serializable. No serial execution may yield an element not previously pushed to the queue



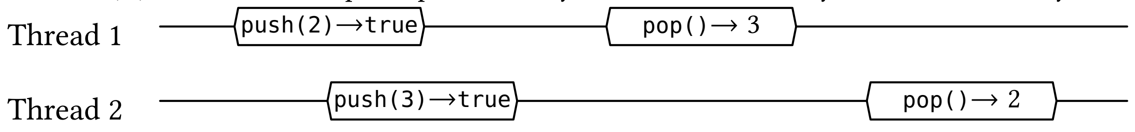
(b) Serializable, but not inter-thread serializable. Can only be serialized by reordering the push operations which requires reordering operations within thread 1



(c) Inter-thread serializable, but not linearizable. The push (or pop) operations may be reordered as they execute in different threads.



(d) Linearizable. The push operations may be reordered since they execute concurrently



3. Related Work

Strict queues. An early lock-free queue design envisioned by Michael and Scott (MS) [10] simply uses a singly-linked list wherein each node stores a modification counter next to the element which in combination with `compare-and-swap` (CAS) allows for avoiding the ABA problem[1].

The Baskets Queue [6] features a similar design to many relaxed queues while remaining completely linearizable. Based on the MS queue it uses baskets as nodes in which multiple elements that have been enqueued concurrently are stored in an unordered manner while the baskets themselves remain ordered via a linked list.

Besides the unbounded linked list approaches there exists a family of concurrent queue designs such as the one presented by Shann et al. [14] that trade the unlimited size for no dynamic memory allocations and better locality of access by using a fixed-size array as a ring buffer, thus improving performance. Additionally, these approaches may further improve performance by using `fetch-and-add` (FAA) over CAS. While CAS is generally faster, it may fail and combined with the cost of the necessary retries it is outperformed by the generally slower, but never-failing FAA at some level of contention [11].

A state-of-the-art non-relaxed design is the LCRQ (List of Concurrent Ring Queues) [11] which combines the ring buffer based approaches with the MS-style linked list to remain unbounded while benefiting from increased performance due to the ring buffers within which most operations occur. It pays peculiar attention to utilizing FAA over CAS and demonstrates that this allows a speedup of 4×–6×. The deficit of this design is that it requires a 2-word CAS operation (CAS2) which is not available in most languages and architectures.

LPRQ (the P standing for portable) [13] alleviates this issue by eliminating the CAS2 only using standard CAS and FAA operations and presents a state-of-the-art strict queue.

Relaxed queues. The concept of quasi-linearizability is introduced along with the Segmented Queue [1]. Quasi-linearizability formalizes the semantics of relaxed data structures by defining it as a bounded "distance" a set of operations may have to a strict execution of the data structure. This distance is referred to as the quasi-factor which is roughly equivalent to the maximum rank error. The Segmented Queue is a singly linked list of segments within which the elements are considered to have the same rank and may be removed (and inserted) in any order.

[1]The problem that an update to a value might be missed if it is changed from A to B and then back to A

The primary issue with the Segmented Queue is that its pop operation might fail erroneously, potentially leading to incorrect early algorithm termination in cases where termination depends on the queue becoming empty. This issue was recognized and rectified by Kirsch et al. with their k-FIFOs [8]. They emphasize the importance of linearizable fullness and emptiness checks in push and pop operations to prevent issues with early termination and propose two variations of the k-FIFO. One is unbounded in size and builds on the Segemented Queue, making its emptiness check linearizable, as well as introducing further optimizations. The other, bounded size, variant exhibits better performance and is based on a circular buffer divided into so called k-segments, segments of size k, which are either completely filled or emptied until the head and tail are advanced. Within a k-segment a random starting position is selected and a linear search for desired elements or slots executed. We will solely refer to the bounded size k-FIFO from this point forward, as it is superior in performance and our design will similarly be bounded.

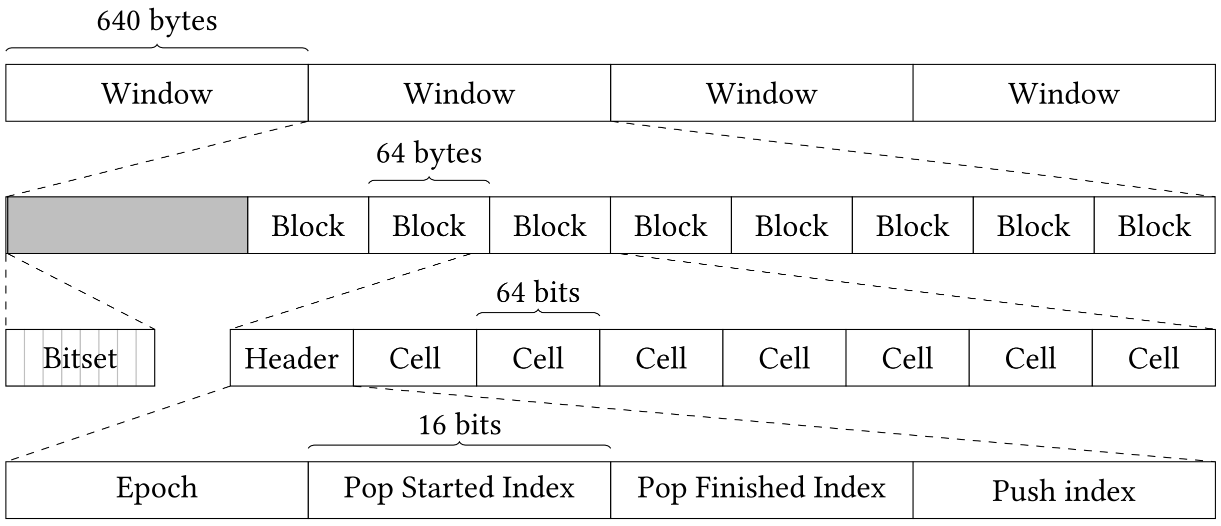
The MultiQueue [15] is a state-of-the-art relaxed concurrent *priority* queue. It uses at least two sequential priority queues per thread, which allows it to utilize wait-free locking. To insert elements it simply selects a random queue, while to remove them two random queues are selected and the minimal element among both of their minimal elements is selected, increasing quality. It uses stickiness as a tuning parameter, which decides how long a thread will execute further operations on the same set of queues, increasing locality of access and thus performance while degrading quality. The MultiQueue can be adapted into a "MultiFIFO" by replacing the internal sequential priority queues with circular buffers and adding a timestamp to each pushed element in order to pick the best among the two ring buffers in pop operations.

Our contribution. We present the **BBQ**, the *block-based queue*, which is, similarly to the k-FIFO, based on a single ring buffer in contiguous memory. It introduces further subdivision, having a three layered structure to allow for sharing less data between threads, benefiting cache efficiency. We insert a bitset into our first-level segments as an acceleration structure to allow for finding full or empty blocks faster. This is intended to allow for higher degrees of relaxation without degradation in performance, as Kirsch et al. explicitly mention this performance degradation associated with high k due to the linear search for the few remaining elements within a large, mostly empty k buffer (which we do the functional equivalent of when searching for bits within our bitset).

4. Block-Based Queue

Our queue is based on a circular **buffer** that we partition into **windows**. Each window is further partitioned into **blocks** as well as an atomic **bitset** featuring one bit per block. Each block has a **header** which is a single 64-bit atomic integer as well as atomic **cells** initialized to 0 to store the actual elements. Figure 4.1 visualizes the layout of a BBQ.

Figure 4.1.: Our to-scale data layout for an example configuration with 4 windows, 8 blocks per window and 7 64-bit cells per block, totaling 2560 bytes. Note the padding on the single-byte bitset to assure 128-byte alignment so that bitset and blocks do not share a cache line



Epoch. Due to the nature of circular buffers we need to distinguish between the same physical window of the buffer at different points in time and to do this we define an epoch as $\left\lfloor \frac{\texttt{window\_index}}{\texttt{window\_count}} \right\rfloor$ (division without remainder). We only ever use epochs implicitly via the window index in our implementation, as this allows us to store less data and avoid some additional calculations.

Headers. To determine which epoch a block belongs to the header stores its associated window's index. The cells within a block can be seen as a miniature queue and the read and write indices within the header are used to determine its bounds. The second "read finished" index is used to coordinate the full emptying of a block, as pop operations may complete out-of-order so the pop operation popping the last element might finish while other pop operations are still ongoing within the block.

Bitset. To determine whether a block contains any elements the bit of its index within the window is set or unset within the window's bitset. For short periods of time a bit may be set when the block contains no elements, but the bit may never not be set when the block contains elements.

Configuration. The BBQ can be configured via two parameters:

1. The amount of *blocks per window per thread* henceforth *block factor B*
2. The amount of *cells per block C*

A `4-63-bbq` is our BBQ configured with B=4 and C=63. We later apply a similar naming scheme to our competitors as well.

A rough sketch of both push and pop operations can be found in Figure 4.2.

4.1. Inner-Block Operations

Blocks are essentially small, reusable, limited-size concurrent FIFOs with strict semantics. Within a block there exist three relevant variables with one invariant:

$$\text{read started index} \leq \text{read finished index} \leq \text{write index} \tag{4.1}$$

The write and read started indices are used in push and pop operations respectively to claim a cell. To do this and to start the operation they increment them atomically, the previous value being the index of the cell that was successfully claimed.

Push. A push operation concludes by atomically replacing the previous value of its claimed cell (which must be 0) by the pushed value.

Pop. A pop operation continuously reads the value stored in the cell until it is not 0, replaces it with a 0 (to remain consistent for the next write epoch) and concludes the operation by incrementing the read finished index. It is also responsible for resetting the block for use with the next epoch if the read finished index after its execution equals the write index. This reset involves a CAS operation on the header meaning if an additional element was pushed in the meantime (involving the increment of the write index), the CAS can just fail and the pop operation concludes normally. If the CAS concludes successfully the block has been reset and it must additionally be marked as empty in the bitset.

Pseudocode for both inner block operations can be found in Figure 4.3.

In general, all inner-block operations support both multiple readers and multiple writers (even at the same time) by splitting operations into a "claim" step on the header and a "commit" step executed on the cells themselves. Furthermore, there have to be two read indices, as multiple readers need to coordinate in the resetting of the block, which must be

Figure 4.2.: Rough sketch of push and pop operations

Algorithm 3: Push

```
1. Is the current block valid for pushing? That is to say, is its epoch equivalent to ours
   and are there free cells as indicated by the push index? If so, go to step 4.
2. Have we previously claimed a new block in this algorithm's execution (spurious
   claim)? If so, release that block by resetting its filled bit within the window's bitset.
3. Try to claim a new block. If it succeeds, go to step 1; otherwise, there are no blocks to
   push to, so cancel the operation.
4. Try to increment the header's push index atomically via CAS. If it fails, go to step 1.
5. Insert the element in the cell of the push index' value before the CAS.
```

Algorithm 4: Pop

```
1. Is the current block valid for popping? That is to say, is its epoch equivalent to ours
   and are there cells containing elements as indicated by the push index being larger
   than the pop index? If so, go to step 3.
2. Try to claim a new block. If it succeeds, go to step 1; otherwise, there are no blocks to
   pop from, so cancel the operation.
3. Try to increment the header's pop index atomically via CAS. If it fails, go to step 1.
4. Wait until the element in the cell of the pop index' value before the CAS is not 0.
5. Swap the element in the cell of the pop index' value before the CAS with 0.
6. Increase the read finished index atomically via fetch-and-add.
7. Is the read finished index larger or equivalent to the push index? If not, stop.
8. Increase the block's epoch and reset all indices. Additionally, unset the block's filled
   bit in the window.
```

done by the reader last finishing their read operation, which is not necessarily the reader reading the last cell, as read operations might complete out of order.

The pop operation contains the singular lock within our implementation and it occurs between the claim and commit steps in the push operation. The lock can only occur in a non-full block in which both a reader and a writer are present. This lock cannot be avoided in the current implementation, as the lock occurring in the last remaining block of a window would result in there being no alternative way of achieving progress besides moving the window which may not be done until it is fully emptied.

Figure 4.3.: Inner block operations. This code assumes that the block has already been checked to be valid and will not become invalid during the operation. In the implementation the index updates are combined with the assertion for block validity in a single CAS

```
Algorithm 5: PushInner
Input: T $value \neq 0$
1 $index \leftarrow write\_index{++}$;
2 $cells[index] \leftarrow value$;
```

```
Algorithm 6: PopInner
Output: T $value \neq 0$
1 $index \leftarrow read\_index{++}$;
2 repeat
      // Noop
3 until $cells[index] \neq 0$;
4 $value \leftarrow cells[index]$;
5 $cells[index] \leftarrow 0$;
6 if $++read\_finished\_index = write\_index$ then
7     if try resetting header via CAS then
8         reset filled bit in bitset;
9 return $value$
```

4.2. Block Claiming and Window Moving

Claiming a new block for reading and writing starts by inspecting the bitset of the current respective window for a block of interest, either one that is marked as filled or not filled respectively. If one such block is found then it is either atomically marked as filled in the case of a writer or remains marked as filled in the case of a reader, which concludes the block claim operation. The distinction between mutating and non-mutating bit claiming enables multiple readers to share a block, while each writer has exclusive write access to its block, although concurrent readers may be present. This distinction is sensible, as a writer falling dormant while claiming a block is of no consequences, as it will be emptied and reset as usual once the read window sufficiently advances, while a reader falling dormant in the same situation would lead to elements being left behind were access to the reader's block exclusive, which could eventually lead to a locking up of the queue when the write window reaches the previous read window in the next epoch and neither window can advance due to the remaining elements in the read block. See Figure 4.4 for pseudocode.

If claiming a bit fails that either means all blocks within the window are occupied by a writer in the case of a write block claim or that all blocks are completely emptied in the case of a read block claim. This necessitates moving the window, which is rather simple and the pseudocode of which can be found in Figure 4.5.

Figure 4.4.: Pseudocode for claiming a block, applicable to both read and write blocks

```
Algorithm 7: ClaimBlock
Output: bool success
1  loop
2      curr_window ← Load(window);
3      bit_index ← ClaimBit(curr_window.bitset);
4      if bit_index is valid then
           // Update thread local block
5          my_block ← curr_window.blocks[bit_index];
6          return true;
7      else
8          if MoveWindow(curr_window) then
               // Reloaded the window, try again with new window
9          else
10             return false;
```

Figure 4.5.: Pseudocode for moving the window, generally applicable to both read and write windows

```
Algorithm 8: MoveWindow
Input: curr_window
Output: bool success
1  if other window is not too close then
2      CAS(window, curr_window, curr_window + 1);
       // If CAS fails it was moved by a different thread
3      return true;
4  else
5      if we're moving the read window $\land$ current write window contains elements then
6          ForceMoveWriteWindow();
7          return MoveWindow(curr_window);
8      else
9          return false;
```

There exist two special cases that are somewhat related in which close attention must be paid in order to preserve the semantics of the data structure, these are outlined below:

Force-moving the write window. When the read window cannot be moved because it is right behind the write window, but the write window contains elements as indicated by the bitset[1], returning `false` would be incorrect for the pop operation, as there *are* elements

[1]It is irrelevant if this check only succeeds because of a spurious claim, as in that case we will simply force-move the write window anyways and move onto an empty window, from which we will immediately try to move on further

within the queue. For this reason we move the write window in this situation to prevent both windows from interfering, which they would do because of the bitset not being able to be accurate for both windows at once (either there are blocks marked as filled that are actually empty that readers would try to claim, or there are blocks marked as empty that are already sealed for the next epoch which writers would try to claim). However, force-moving the write window requires special attention in order to not render blocks unusable due to their epoch falling behind (which cannot be remedied later, as we need to rely on epochs being exact and not simply "smaller" due to eventual integer overflows). Without force-moving the window, the write window does not move until all blocks have been claimed, meaning they each contain at least one element. This means that every block will be properly emptied and then reset after the read window has reached it. This is not the case for force-moving the write window, in that case there might still exist blocks which have not been claimed by any writers, which means the reader that is responsible for force-moving the write window must also assure that all blocks properly get reset for the next epoch. To do this, it must simply manually reset all blocks within the current window that do not contain any elements. This must occur strictly before moving the write window, as it needs to be done before the read window is allowed to move onto the previous write window. If the write window were moved first, the next reader that tries to claim a new block sees that the windows are not next to each other and simply moves the read window without being aware of the force-move in progress. This allows for both windows to potentially continue moving, even though not all blocks have yet been reset. In this specific situation it would be possible for a writer to successfully claim a block within the old write window, even though the read window is now beyond that point, leading to that block becoming stuck with an old epoch and its elements lost (because the read window has already moved past and the reader force-moving the write window can't reset blocks that contain elements). When force-moving, the reader must not rely on the bitset and needs to check manually whether each block is not yet reset and empty, because of the potential for a spurious claim (see below). All blocks that are not empty will be emptied and reset by the readers as normal after moving the read window.

Spurious claim. Normally, a reader is responsible for resetting the filled bit of a block when it has been emptied completely, however, when a write window is being forced-moved, blocks can be reset despite not having contained any elements in this epoch. In such a case it is possible for a writer to be in the process of claiming a block, setting the bit in the bitset, even though the block has already been sealed from the current epoch, meaning it will not be able to push to it. In such a case it is solely responsible for resetting the bit in the bitset upon claiming the next block, as it alone is capable of identifying this situation with certainty. If each reader encountering a block that has its bit set but containing no elements were to reset its bit, this could result in situations where a writer was in the process of writing the blocks first element, having the bit set already, when the reader would see that the block is empty, resetting the bit after which the writer would start filling the block, even though its bit is no longer set. This must never happen. Spurious claims themselves are no issue, as readers encountering a block marked as filled in the bitset but without any elements simply try to claim a new block, as it is either a block that is about to be filled or a spurious claim.

4.3. Explicit Emptiness and Fullness Checks

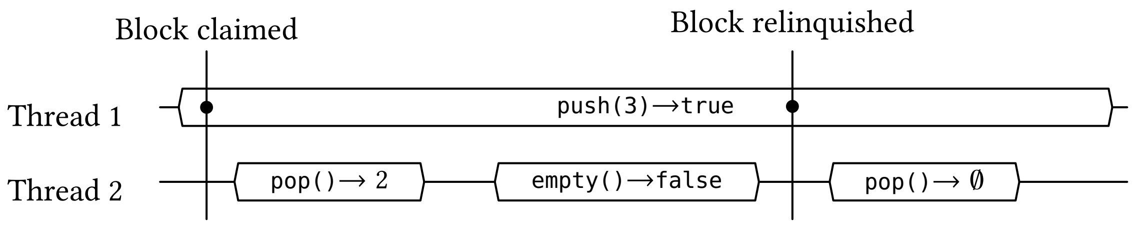
At first glance it appears emptiness and fullness checks may easily be implemented as degenerate pop and push operations respectively by replacing all mutating operations with their non-mutating counterpart, collapsing and simplifying the result and adding additional conditions to take into account changes to the data structure that may occur during the execution of the checks. Care must be taken when implementing the additional conditions as they are implicitly checked by the mutating operations and if care is not taken the non-mutating ones may lead to an incorrect result.

Emptiness Check. In the case of the emptiness check care must be taken to actually make sure blocks marked as "filled" contain any elements, as otherwise in the case of a spurious claim there exists a series of operations that results in behavior that is not inter-thread serializable. As a precondition for this situation to occur the write window must be adjacent to the read window, there must be exactly one element within the current write window and no elements within the current read window.

1. Thread 1 starts a push operation and marks a block B in the write window as filled without yet increasing its write index.
2. Thread 2 conducts a pop operation. It needs to force-move the write window and thus increases the epoch counter in all remaining blocks within the current write window without any elements present (including B) before increasing the write window and popping the singular element within the now-former write window. (The queue is now empty, save for the spurious block claim by thread 1.)
3. Thread 2 conducts an emptiness check, which sees the claimed block in the current read window (former write window), returning `false`.
4. Thread 1 tries to increment the write index but fails due to the updated epoch, releases the block claim and starts looking to claim a new block in the new write window.
5. Thread 2 conducts a pop operation that fails, as no blocks are marked as filled.
6. Thread 1 concludes its push operation.

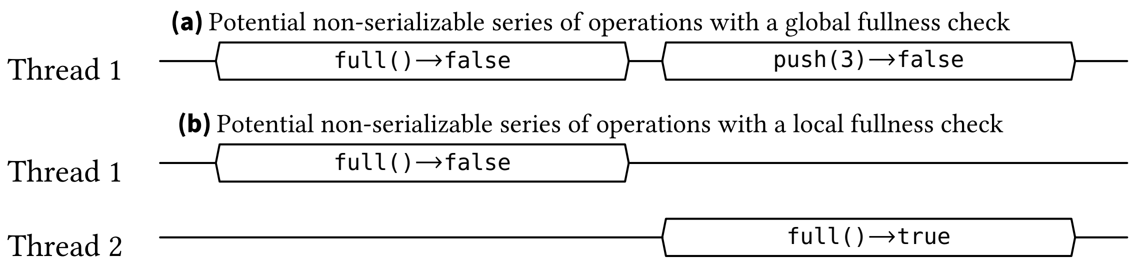
See Figure 4.6 for a visualization of the required concurrency. It is serializable by ordering the failing pop operation before the emptiness check and the push operation in between, but not inter-thread serializable (and thus not linearizable) as this serialization requires reordering of operations within thread 2.

Figure 4.6.: Concurrency diagram of a series of operations with an incorrect emptiness check that cannot be inter-thread serialized



Fullness Check. There exist two possible approaches that both suffer from semantics that seem inconsistent from a certain perspective. This is caused by the exclusive ownership of write blocks, which itself leads to behavior that could be considered inconsistent. In particular: a thread 1 may repeatedly report not being able to push an element to the queue, while a thread 2 repeatedly being able to do so (up to $C - 1$ times). This situation can occur when the write and read windows are adjacent and thread 1 has no claimed block, while 2 has a claimed block that still contains free cells (although at least one filled cell as blocks are immediately pushed to upon being claimed). So a check for fullness could either check whether *any* thread is able to push (global fullness) or whether only the calling thread is able to push (local fullness), the latter replicating the problem described above while the former can lead to a situation where a thread can repeatedly not push despite the fullness check returning false. Neither of these implementations would be serializable, as demonstrated in Figure 4.7.

Figure 4.7.: Concurrency diagrams of the semantic issues affecting different approaches to implementing a fullness check


(a) Potential non-serializable series of operations with a global fullness check
Thread 1
full()→false
push(3)→false
(b) Potential non-serializable series of operations with a local fullness check
Thread 1
full()→false
Thread 2
full()→true

To allow a fully linearizable fullness check block ownership must be non-exclusive, at least in a scenario where a push operation might otherwise fail (almost full queue). In general a fullness check is not as relevant as an emptiness check anyways, as it cannot reasonably be used for termination.

4.4. Atomic Bitset

The bitset is an integral part of the BBQ's design. It is both a hot spot as identified by flame graphing as well as a significant contributor to the BBQ's performance as compared with a variant that only implicitly stores the filled state of blocks via the header's push indices.

Our N-bit bitset consists of L elements of the bitset's internal type of size I. For simplicity in implementation N is restricted to multiples of the internal type's size.

There are three primary operations that our bitset needs to support:

4.4.1. Emptiness check

`any` allows for checking whether the bitset has any of its bits set, to this end it simply iterates over all internal elements and compares them against `0`, returning true if the comparison fails. This check is used exactly once during read block claiming when the read window immediately precedes the write window and in order to decide whether the current write window contains any elements the `any` function is called on its bitset. This decides between the pop operation failing if there are no filled blocks or the read window being moved following the write window being force-moved if there are.

4.4.2. Setting a Bit

`set` and `reset` allow modifying a single bit and return whether the mutation actually had an effect. They are implemented using `fetch_or` and `fetch_and` respectively, which atomically apply a bitmask to an atomic and return the atomic's value before the mutation. See Figure 4.8 for pseudocode for `set`.

`reset` is used when releasing spuriously claimed blocks and when resetting a block after emptying it.

Figure 4.8.: Pseudocode for the atomic bitset's `set`

```
Algorithm 9: Set
Data: Atomic<T> data
Input: bit_index
Output: true if the bit was flipped in this operation; false if it had already been set
// If the returned value already has the bit set no mutation took place
return $data.fetch\_or(1 \ll bit\_index) \,\&\, (1 \ll bit\_index) = 0$
```

4.4.3. Bit Claiming

`claim_bit` is the most important bitset operation by peformance impact. It looks for either a 0 or 1 within the bitset and either simply returns its index or atomically flips it before returning its index. This behavior can be controlled using two boolean parameters, but only two parameter combinations are in use. One seeks to find a 0 in the bitset, atomically flips it to a 1 and the other seeks to find a 1 but does not automatically flip it.

They are used when looking for an empty block to fill or a full block to empty respectively. We only present pseudocode for the first variant in Figure **??**, as the modifications required to attain the second variant are trivial.

Figure 4.9.: Pseudocode for the atomic bitset's `claim_bit`

```
Algorithm 10: ClaimBit
Data: Bit Count $N$; Internal Element Count $L$; Internal Element Size $I$
Output: Index for a bit that was flipped from 0 to 1 or -1 if all bits are already 1
1  $offset_o \leftarrow$ Rand($L$); // Outer offset (within buffer)
2  $offset_i \leftarrow$ Rand($I$); // Inner offset (within internal type)
3  for $i \leftarrow 0$ to $L$ do
4      $index_o \leftarrow (i + offset_o) \% L$;
5      $raw \leftarrow$ Load($buffer[index_o]$);
6      loop
7          $rotated \leftarrow$ RotateRight($raw, offset_i$);
8          $c \leftarrow$ CountRightOne($rotated$);
9          if $c = S$ then
10             break;
11         $index_i \leftarrow (offset_i + j + c) \% I$;
12         loop
13             $test \leftarrow raw \mid (1 \ll index_i)$;
14             if $test = raw$ then
15                 break; // The bit has already been set in a different thread
16             else if CAS($buffer[index_o], raw, test$) then
17                 return $index_i$; // We managed to change the bit
18             else
                   // The CAS has updated raw with the current value, retry
19 return $-1$;
```

To avoid contention on singular bits we want to spread out the bits we're claiming as much as possible, which is why we're working with a random starting offset that we generate using a cheap thread-local `std::minstd_rand` generator[2]. We utilize the C++20 `std::countr_one` and `std::rotr` functions which are intended to compile to a single hardware instruction.

[2]The difference in overall performance to `std::ranlux24_base` was not statistically significant

`std::countr_one` allows us to iterate directly over the bits we are looking for and `std::rotr` enables us to integrate the initial offset. This leads to a noticeable speedup over more naive approaches.

4.5. Implementation Details

Portability. We restrict ourselves to purely standard C++20, avoiding both platform and compiler-specific functionality as well as inline assembly in order to maximize portability in contrast to all publicly available competitors. This allows our implementation to be used both on Microsoft Windows (see Figure A.4) as well as ARM-based processors (see Figure A.3).

Modulo Optimizations. The window count and the amount of blocks per window is automatically rounded to the next power of two, as this allows for optimizing modulo operations as a simple bitmask. This benefits us within the bitset and when calculating the physical block index within the queue itself. Additionally, there must be at least 3 windows for a working implementation (so 4 due to the power of two requirement) as well as at least as many blocks within a window as there are bits within the internal type of the bitset.

Handles. The only way to access operations on the queue is via handles that store non-static thread-local data. They store a pointer to the thread's read and write blocks and the associated window indices for reading and writing. Handles are not thread safe. The necessity for handles is a downside to our approach.

Cache Alignment. We seek to align all mutable data within our implementation that is concurrently accessed by multiple threads to the size of a cache line to prevent false sharing. This applies to the read and write windows within the queue itself, as well as the bitset and block array within a window. We do explicitly not align the blocks themselves or the data within, as even on the lowest sensible configuration they are already multiples of 64 bytes large (1×8 byte header + 7×8 byte cells) and the true sharing within the blocks is a major contribution to the BBQ's performance.

4.6. Analysis

- W is the window count.
- B is the amount of blocks within a window.
- C is the amount of cells within a block.
- I is the size of the bitset's internal type in bits.
- T is the size of the element type in bytes.

- CS is the size of a cache line in bytes.

Given the the user-configured amount of blocks per window BI, then

$$B = \min\{x \mid 2^n = x,\ n \in \mathbb{N},\ x \geq \max\{BI, I\}\} \tag{4.2}$$

That is to say, the actual block amount is the smallest multiple of the internal bitset's type (so there are no "unused" bits in the bitset) that is also a power of two (to optimize modulo operations).

Given the user-configured desired minimum capacity of the queue S, then

$$W = \max\left\{4, \min\left\{x \;\middle|\; x = 2^n,\ n \in \mathbb{N},\ x \geq \frac{S}{B \cdot C}\right\}\right\} \tag{4.3}$$

That is to say, the actual window count is the smallest power of two that is greater than 4 and able to store all the cells.

4.6.1. Time

For our runtime analysis we focus on the expected runtime for low contention scenarios, as a heavily contented CAS might never succeed.

We start from the bottom up. Our bitset's `set`/`reset` functions are $O(1)$, as they execute a single atomic fetch-modify operation. The `any` function is $O\left(\frac{B}{I}\right)$ as it needs to iterate over all $\frac{B}{I}$ elements in the bitset's array. `claim_bit` takes at most $\frac{I}{2}$ iterations of the `countr-rotr` routine in the case of an alternating bit-pattern and needs to repeat this for at most $\frac{B}{I}$ elements, leading to $O(B)$.

Force-moving the write window takes $O(B)$, needing to iterate over all blocks in the window. This means that in general moving a write window is possible in $O(1)$, while a read window requires $O(B)$. However, each block claim, which may or may not involve the moving of a window, necessarily incurs the runtime of claiming a bit from the bitset, resulting in $O(B)$ for both claiming a new block and pushing as well as popping an element, as neither of the additional operations within those functions add any additional complexity.

4.6.2. Space

For the actual data within the BBQ the following space is required: A block requires $BS := C \cdot T + 8 \overset{T=8}{=} (C+1) \cdot T$. Thus, the space requirements for a window are $WS := B \cdot BS + \frac{B}{8}$, the second addend resulting from the bitset storing B bits. Accounting for padding and alignment this gives us $WS_a := CS \cdot \left(B \cdot \left\lceil \frac{BS}{CS} \right\rceil + \frac{B}{I}\right)$, as block sizes are rounded up to the next multiple of the cache line size and each internal element within the bitset is equally placed on its own own cache line. Finally, $W \cdot WS_a$ or $W \cdot WS$ gives us the total size of our data block either with or without consideration of alignment.

The data structure itself stores 5×64-bits in total[3], 2 for the window count and its modulo bitmask, 1 for the pointer to the actual data and 2 for the read and write window respectively. Only the last two are mutable and thus aligned to cache lines, giving us a total size of $3CS$, assuming $CS \geq 3 \cdot 8$ bytes.

Handles store 4×64-bits in total, pointers to their claimed read and write blocks and their associated epochs for both. They do not benefit from additional alignment as they are not shared between threads, giving us a size of $4 \cdot 8$ bytes.

4.6.3. Quality

Our maximum rank error within a window is $(B-1) \cdot C - 1$. The $B-1$ over B stems from the fact that within each block strict per-thread FIFO semantics apply, meaning the first element popped from a block is also the one first pushed to it, so even if the "worst" block is selected, within it we pop the best element first. This aspect of the BBQ is hard to quantify but might have a genuine positive effect on its characteristics, as within each block you receive *all* elements in the correct order relative to one another, although this does not imply they are adjacent in the globally correct order.

In continuous execution (i.e. without any threads (or at least writers) remaining dormant for prolonged periods of time) the order of items between windows is roughly correct, as write and read window are only moved when either all blocks are claimed by a writer or completely emptied respectively. In the case of writing, the window may be moved while there are still writers with a claimed block in the previous block, which must be the case as write access is exclusive. This may lead to an incorrect order between windows.

There exists a pathological case in which the rank error may be as bad as $((W-1) \cdot B + (B-1)) \cdot C - 1$. This can only occur when a writer that claimed a block stops pushing for a prolonged period of time after which it continues pushing although the write window has since long advanced while the read window has not yet reached its block and emptied it. There exists a way to potentially remedy this, see 6.1.

[3]Assuming a 64-bit architecture

5. Evaluation

5.1. Setup

We are using a 64 core x86-64 AMD EPYC 7702 machine with 1014 GiB of RAM running Ubuntu 24.04.1 LTS for our evaluation. We compile using gcc 14.1.0 with the `-O3` flag and utilize core pinning to increase stability and consistency in the benchmarks. We exclusively deal with 64-bit values in our benchmarking. Each benchmark is executed five times and the results averaged, the standard deviation being included via error bars where significant. We use a queue size of $4 \times P^3$ where P is the maximum amount of threads. We have found this to strike a good balance between never coming close to scenarios where the tail encroaches on the head and maintaining a reasonable memory footprint. Unless otherwise stated we prefill the queue to half of its size before benchmarking in order to model realistic usage scenarios and prevent unfavorable conditions, such as the tail encroaching on the head. We are careful to place all handles in each thread's stack memory to prevent potential false sharing.

All experiments are reproducible via the code provided at `https://github.com/Saalvage/block_based_queue`.

5.2. Parameter Tuning

5.2.1. Level of Relaxation

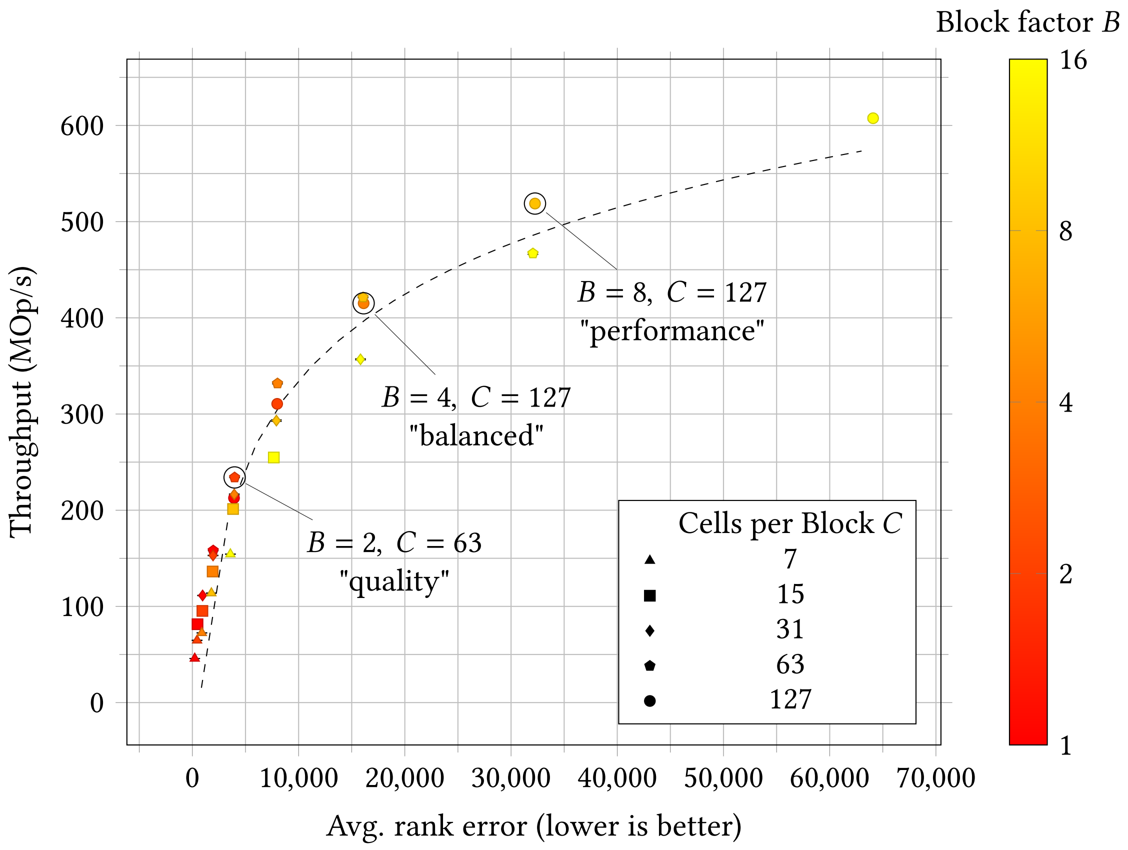
Before evaluating our queue in comparison to others we first compare differently configured versions of it, seeking to find sets of parameters striking a balance between performance (5.4.1) and quality (5.4.2).

We can tune the two parameters described in 4: Configuration, the block factor (blocks per window per thread) B and the cells per block C.

We determine parameter ranges for both parameters, opting to rely on powers of two for both[1]. Blocks per window per thread $\in [1, 16]$ and cells per block $\in [7, 127]$. These values appear sensible as the observed logarithmic growth suggests no more significant improvements to performance beyond this point while the degradation in quality is only accelerating.

[1]Concrete values for cells per block are always $2^x - 1$ due to one "cell" being occupied by the block header

Figure 5.1.: BBQ parameter tuning. The dashed line represents a logarithmic regression with weights linearly inverse to the tuning parameters in order to properly take into account the few high performance, high relaxation data points



We use the parameter "blocks per window per thread" instead of varying the amount of blocks per window directly as it can be reasonably assumed that the performance relates to the amount of blocks each thread has access to within a given window on average instead of the absolute amount of blocks and preliminary experiments have shown that using less blocks than there are threads has detrimental effects on performance.

It is to be expected that increasing either of these parameters results in an increase to performance at the cost of a degradation in quality. The performance increase is logarithmic to the quality decrease, meaning relaxation is only really beneficial up to a certain point.

An interesting observation is the formation of clusters of datapoints, suggesting that a doubling in blocks per window per thread is roughly equivalent to a doubling of cells per block. This suggests the dimensionality of our parameter space can be reduced to a single parameter, representing the amount of cells per window with the division of cells into blocks being largely irrelevant, although there appears to be a slight constant performance benefit to increasing C over B.

Block Size ⇔ Performance. Directly caused by the increase in time spent in cache-friendly inner-block operations and between expensive block-claim operations.

Block Count ⇔ Performance. Likely related to decreased read contention, as the readers can spend a longer time spread out among the blocks before being "cramped" into the few remaining blocks within a window, causing high contention.

Block Size ⇔ Quality. Is not direct, as a single block per window would result in a queue with strict FIFO semantics, but rather by the increase in difference of accessibility between different blocks. Consider a single cell per block, each cell would have the same chance of being selected as the next pop value, but if we increase the amount of cells per blocks, the cells at the end of a block will forcibly be delayed until the block-local tail has advanced to them, despite them potentially (almost guaranteed for some cells with a block multiplier >1) having been pushed earlier than the first element in a different block.

Block Count ⇔ Quality. Similarly, can be explained by the increase in potential candidates for popping.

We have selected three sets of parameters that strike some balance between performance and quality which are marked in Figure 5.1 and with which we will be moving forward. Additionally, the parameter set with $B = 1$ and $C = 7$ will also be kept around as a sensible minimally relaxed baseline [2].

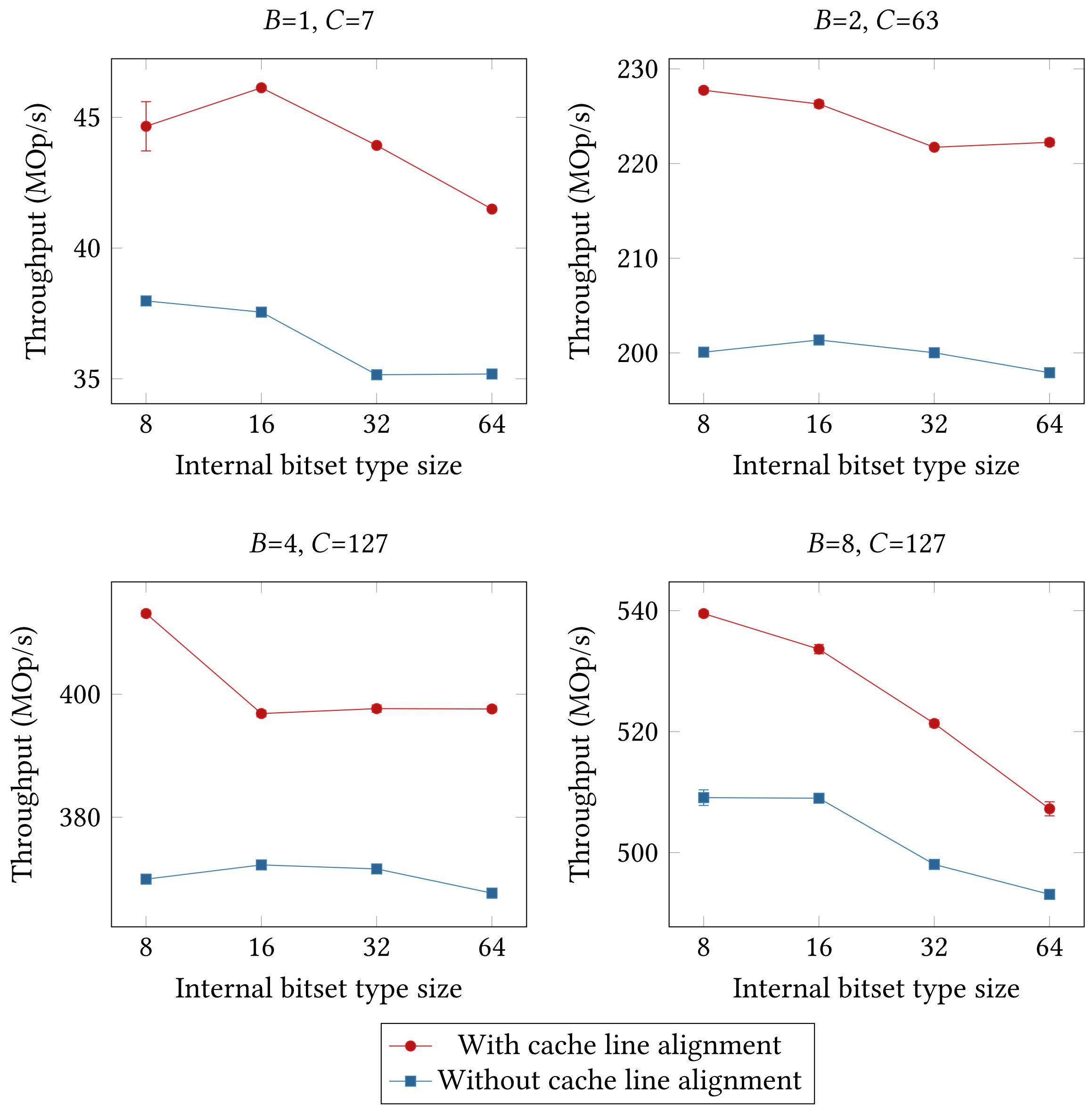
5.2.2. Bitset Internal Type

The internal array type of the bitset can be varied. Additionally, the internal values can be aligned to cache lines. We construct a simple comparison to show that the granularity of the atomic variables is paramount here and that we benefit from cache alignment.

See Figure 5.2 for the comparison.

[2] As well as being the closest competitor to the worse performance, higher quality k-FIFOs, see 5.3

Figure 5.2.: Comparing different bitset types with and without aligning the values to cache lines



5.3. Selecting Competitors

Unfortunately, public implementations of concurrent queues (including relaxed ones) prove quite hard to come by.

We use three sources:

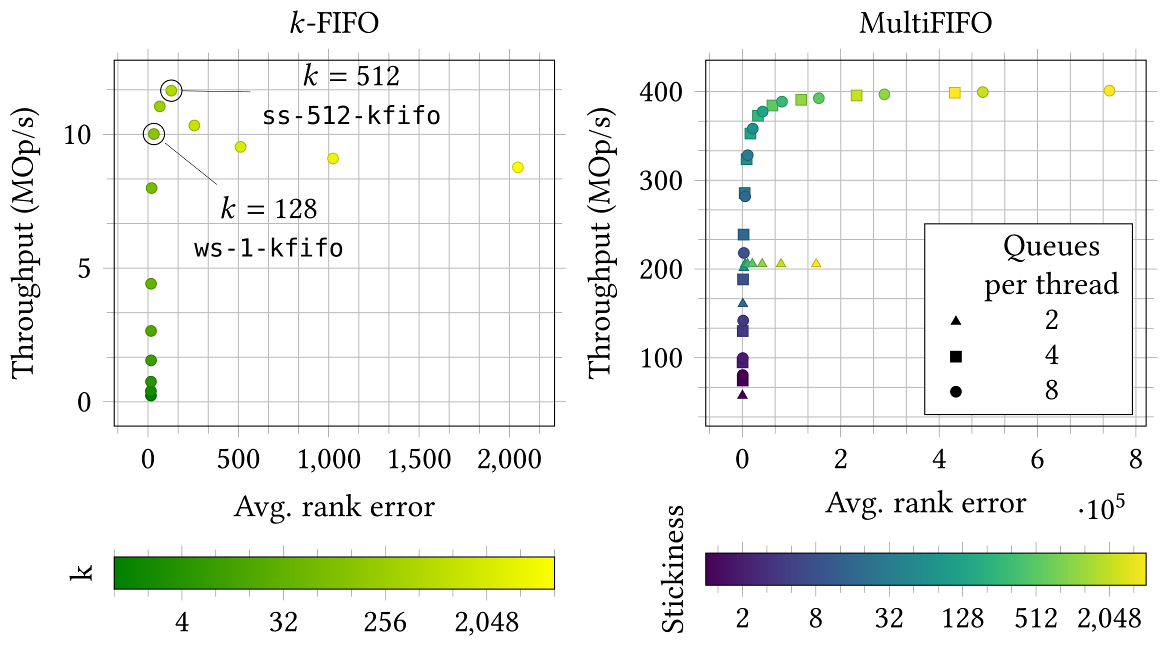
1. `https://zenodo.org/records/7337237` [13]
   Providing high quality, optimized implementations of strict queues from which we have selected the LCRQ alone, as it's the state of the art queue and surprisingly offers

much better, although slightly erratic, performance over the LPRQ as seen in Figure A.2. It will act as a baseline to compare the relaxed approaches to.

2. `https://github.com/cksystemsgroup/scal` [4]
   Providing a variety of both strict and relaxed queues it unfortunately suffers from old age. Many of its implementation are broken and the ones that do work tend to reimplement what is now contained in the C++ standard library in a likely suboptimal way as well as generally suffering from being unoptimized. We use its unbounded k-FIFO implementation in the knowledge that a fully optimized variant is likely to perform better by a constant factor.

3. `https://github.com/marvinwilliams/multiqueue/tree/multififo` [15]
   Provides a FIFO variant of the highly optimized MultiQueue relaxed priority queue. The MultiFIFO uses multiple strict non-concurrent ringbuffers among some of which it selects an element to pop. We configure it using two parameters, Q, the amount of queues per thread and S, the stickiness. See the paragraph about the MultiFIFO in 3 for more details. The amount of queues is exclusively weak-scaled.

Figure 5.3.: Competitor parameter tuning



As can be seen in 5.3, the k-FIFO's performance peaks at a k of 512 and then proceeds to drop off, which is consistent with the observations by Kirsch et al. and explained as "the population in the dequeueing segment [becoming] sparse" [8]. We choose to use two configurations, one as suggested by the authors of setting $k = p$ (weak scaling with k=1, `ws-1-kfifo`) and one at our observed maximum performance at $k = 512$ (`ss-512-kfifo`, for this variation strong scaling has shown to perform better for $p > 4$, see Figure A.1).

Figure 5.4.: Comparing competitors. See Figure 5.1 and Figure 5.3 for reference

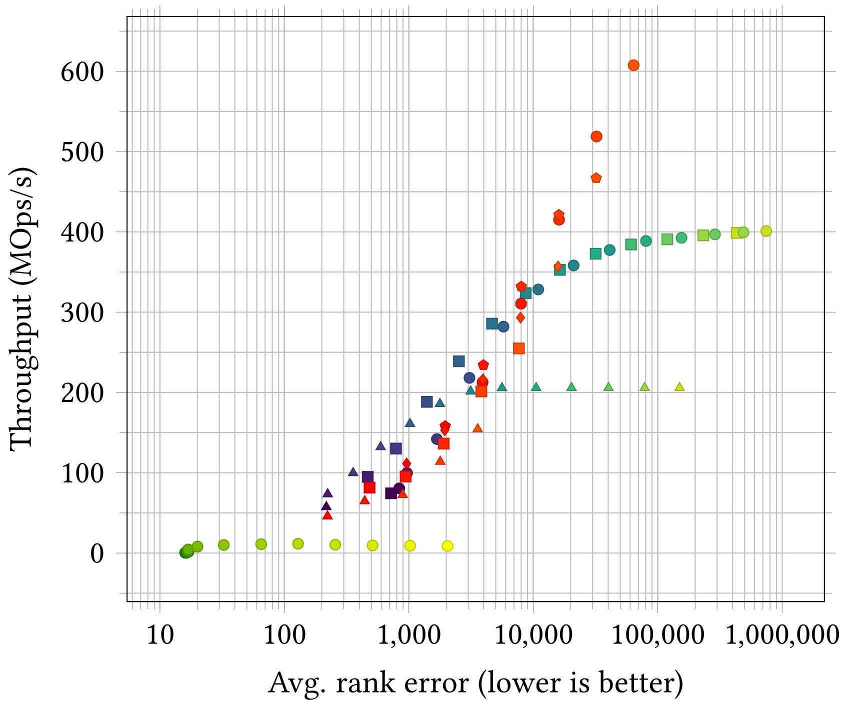
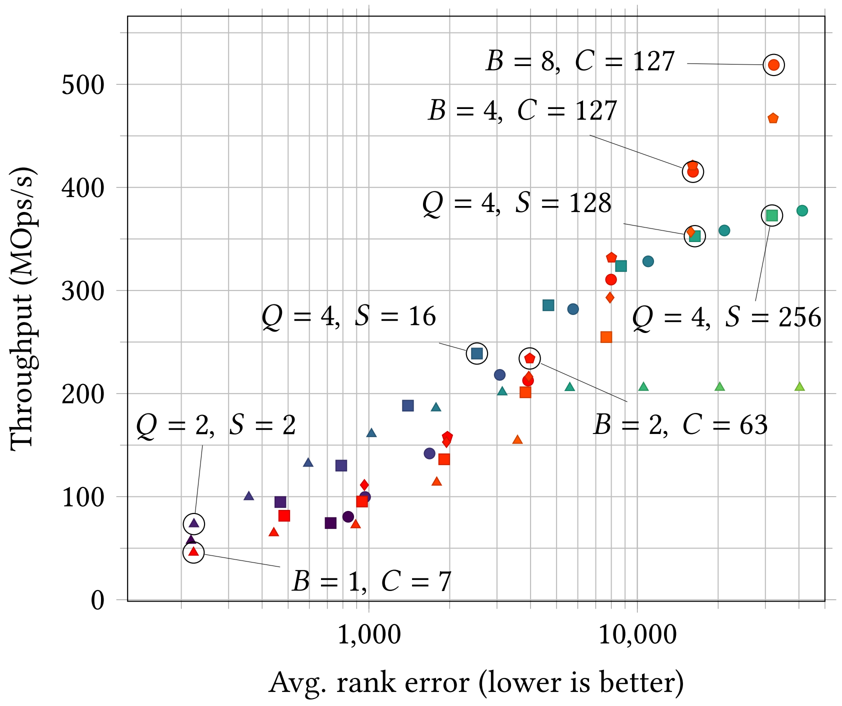


Figure 5.5.: Selecting MultiFIFO parameter sets. Zoomed in version of Figure 5.4



The behavior exhibited by the MultiFIFO is very interesting. In general it plateaus relatively quickly with decreasing quality. Doubling the queues per thread from 2 to 4 almost doubles the achievable performance at high enough levels of relaxation, although it plateaus slightly slower. However, further doubling the queues per thread from 4 to 8 has a minimal *negative* effect on both performance and quality characteristics and primarily only seems to result in a quicker plateauing. We select four competitors based on the performance and quality characteristics of our four sets of parameters for the BBQ as marked in Figure 5.5.

Generally, as can be seen in Figure 5.4, the performance of the k-FIFO fails to reach either of the other implementations, yet its quality is equally unmatched by either. For higher qualities the MultiFIFO offers better performance at a better quality than the BBQ, while there is a turning point at an average rank error of roughly 8000 where the BBQ continues scaling at higher degrees of relaxation while the MultiFIFO starts to plateau. This turning point can also better be observed in the zoomed-in plot in Figure 5.5.

5.4. Benchmarks

We utilize four micro- and one macrobenchmark to compare our implementation to others.

5.4.1. Alternating Push and Pop

In this benchmark we seek to capture the expected "happy case" usage characteristics. Each threads pushes and pops one element from the queue in an alternating manner until a fixed amount of time has passed. The total amount of push/pop pairs all threads managed to accomplish in the given time is recorded as the total throughput.

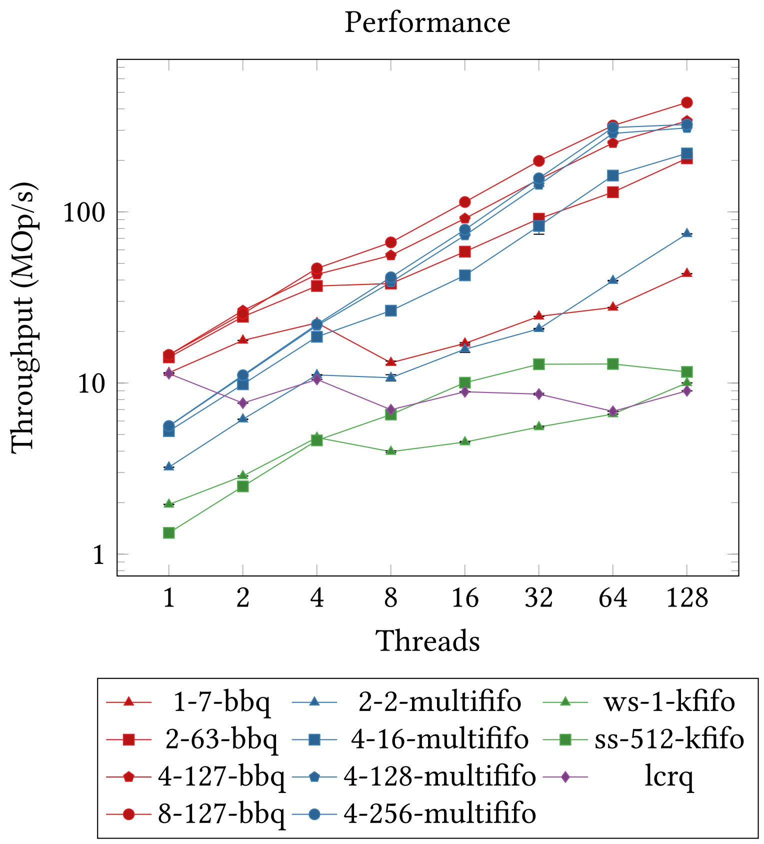
We only consider an artificial case of extreme contention here where there is no local work being done per thread as this is the primary design goal of the BBQ.

Note that since we have directly linked the amount of blocks per window to the thread count this constitutes a form of weak scaling, see 5.4.4 for a benchmark comparing strong scaling characteristics of our BBQ.

As seen in Figure 5.6 the BBQ is able to beat all competing implementation at our chosen highest level of relaxation. The most interesting observation is how the `4-256-multififo`'s performance seems to grow steeply up until a thread count of 64 where it almost reaches the performance of the `8-128-bbq` at which point it abruptly plateaus.

The `1-7-bbq` dropping off at 8 threads is caused by the overallocation of blocks for $p < 8$, as there have to be at least 8 blocks per window to accommodate the bitset.

Figure 5.6.: Performance comparison with competitors. Higher is better



5.4.2. Quality

We push a fixed amount of elements, record push and pop timestamps for each operation and calculate the following quality metrics from them:

Rank error. The difference in push index between the least-recently-pushed element and the popped element. The least-recently-pushed element being popped would have a rank error of 0, the most-recently-pushed element being popped would have a rank error of the entire queue's length−1.

Delay. The amount of times an element within the queue was "delayed" by elements that were pushed later than itself. That is to say, it would've been more semantically correct to pop the element that is being delayed than the element that was actually popped.

Each thread pushes a timestamp into the queue and records a timestamp *after* it has popped an element[3]. This results in a set of timestamp pairs that we evaluate by generating two sorted lists of these pairs, one sorted by push-timestamp and the other by pop-timestamp.

[3]The "after" here is significant as otherwise a case could occur where a pop timestamp lies before a push timestamp.

We then utilize a modified btree to reconstruct the state of the queue at each operation. This allows us to calculate quality metrics for each individual push/pop pair.

Reconstructing the state of the queue for each operation as described above is very expensive and hard to parallelize. For this reason we restrict ourselves to a sample of about 5.000.000 elements per test run.

To gather these samples each thread individually executes push/pop pairs in chunks of 5000 elements, until all threads have collectively completed 1000 chunks. This is coordinated via a global atomic counter counting the amount of completed chunks. The chunking is used to avoid a situation in which individual threads finish much faster than others, leaving less total threads operating on the queue, potentially skewing the results.

Figure 5.7.: Quality comparison. Lower is better. Missing data points exhibit no rank errors

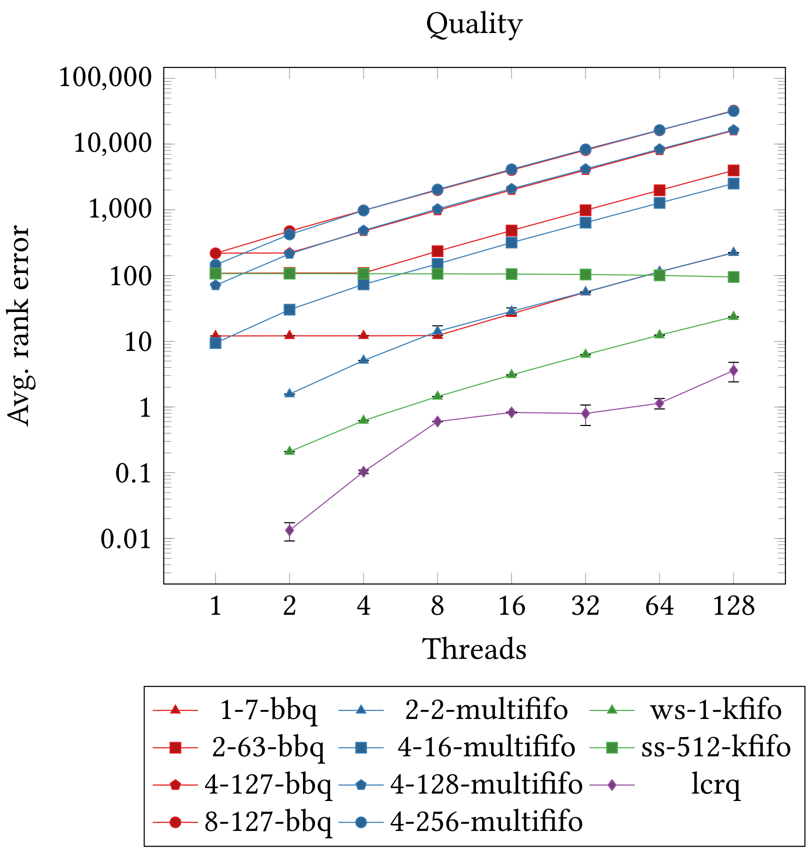


Figure 5.7 demonstrates that quality degrades linearly with thread count for all weakly scaling implementations with the exception of the situation in which the BBQ overallocates the amount of blocks per thread. An interesting observation is that the MultiFIFO's average rank errors for a single thread are always constant, showing that it is deterministic for a single thread. Another interesting observation is that the strong scaling k-FIFO actually improves its quality with higher thread counts. LPRQ performs as expected, as scheduling results in a slight amount of errors even with a non-relaxed queue.

Figure 5.8.: Comparison of the distribution of average rank error and delay between different competitors for $p = 128$. We ignore the most uncommon 1% as it would unnecessarily skew the plot without adding much value

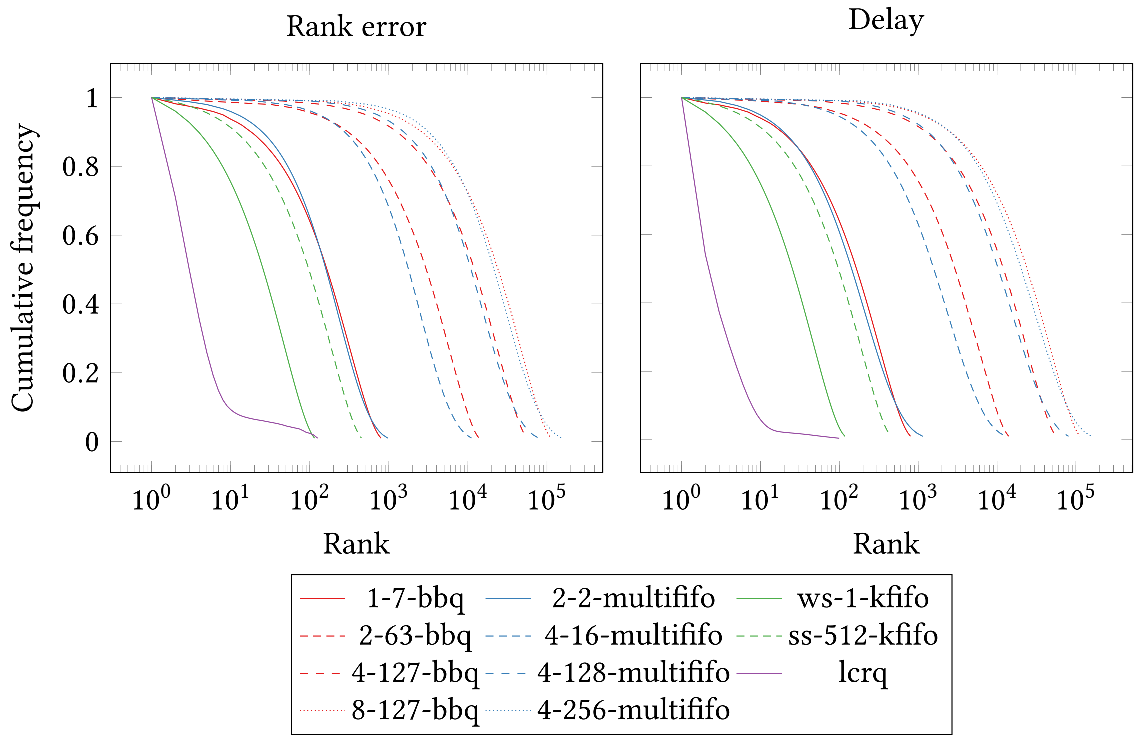
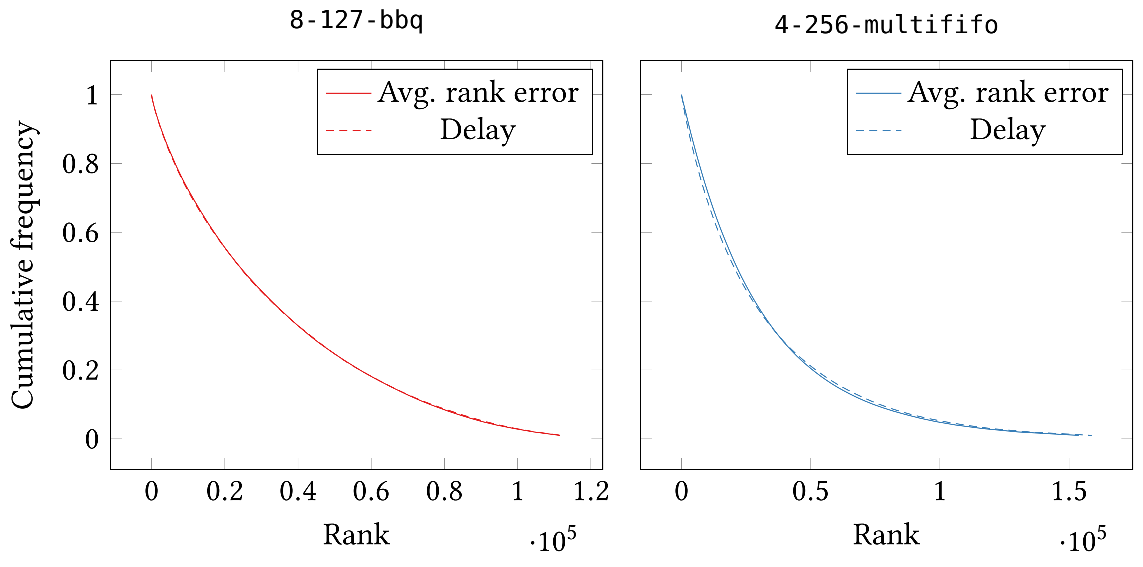


Figure 5.9.: Comparison between average rank error and delay distribution for BBQ and MultiFIFO

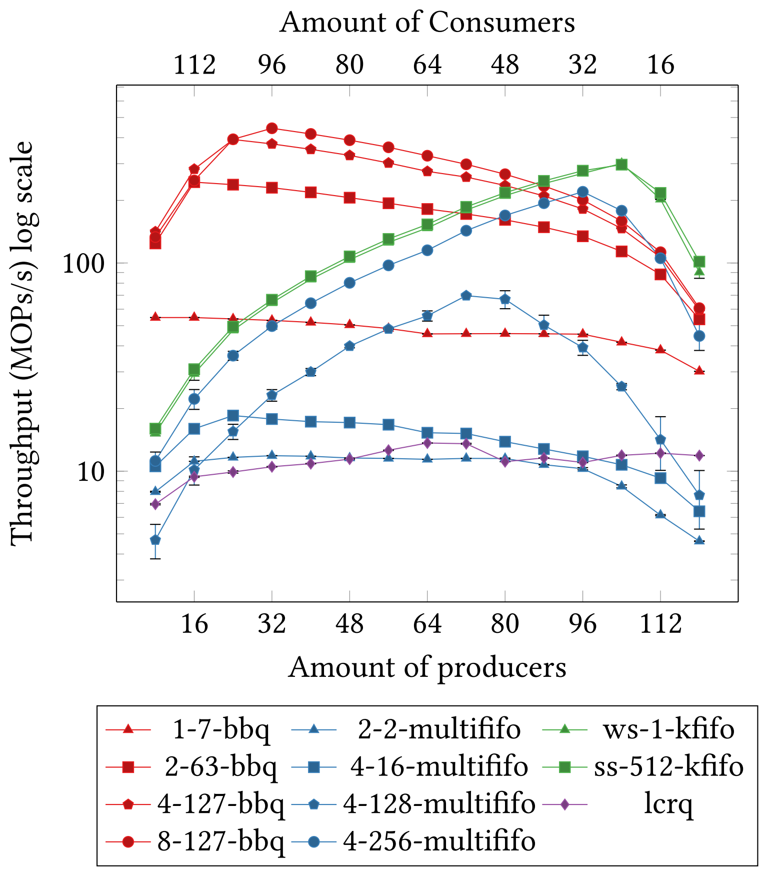


As can be seen in Figure 5.8 and Figure 5.9 there are no significant differences between the different relaxed queues when it comes to the distribution of rank errors and delays.

5.4.3. Producer-Consumer

A given amount of threads continuously pushes elements while the rest continuously pops elements. We vary the amount of each via a predefined ratio. The purpose of the producer consumer benchmark is to find a balance between pushers and poppers where the queue performs optimally. The producer-consumer benchmark can also be used to gauge the difference in performance between push and pop operations, as one would expect a queue with vastly superior push performance to perform better in consumer-heavy benchmarks, although this might result in more contention on the consumers, potentially leading to a general degradation in performance.

Figure 5.10.: Comparing different producer/consumer ratios



As we can see in Figure 5.10, performance does not vary wildly between ratios at lower levels of relaxation, which also goes for the unrelaxed LCRQ.

The BBQ performs a lot better with few producers and many consumers, which may simply be attributed to pushers never experiencing contention in their blocks. This behavior seems to grow more pronounced with increased levels of relaxation, although the sharp dropoff also seems to occur with increasingly more balanced ratios. The most relaxed BBQ performs optimally at a ratio of 1:3 (32:96).

The opposite behavior can be observed with almost all other designs and configurations. The MultiFIFO generally performs very at unbalanced ratios and higher levels of relaxation, having a general preference for more producers that is accentuated at the highest level of relaxation.

The *k*-FIFO performs surprisingly well in this benchmark, squarely beating the MultiFIFO in all tested configurations and also beating the BBQ at the four most extreme producer-heavy ratios. This supports the hypothesis that it is just the alternating push-pop benchmark that it performs exceptionally poorly at. It also has a strong preference for ratios with more producers than consumers and its curve follows the one of the most relaxed MultiFIFO surprisingly closely.

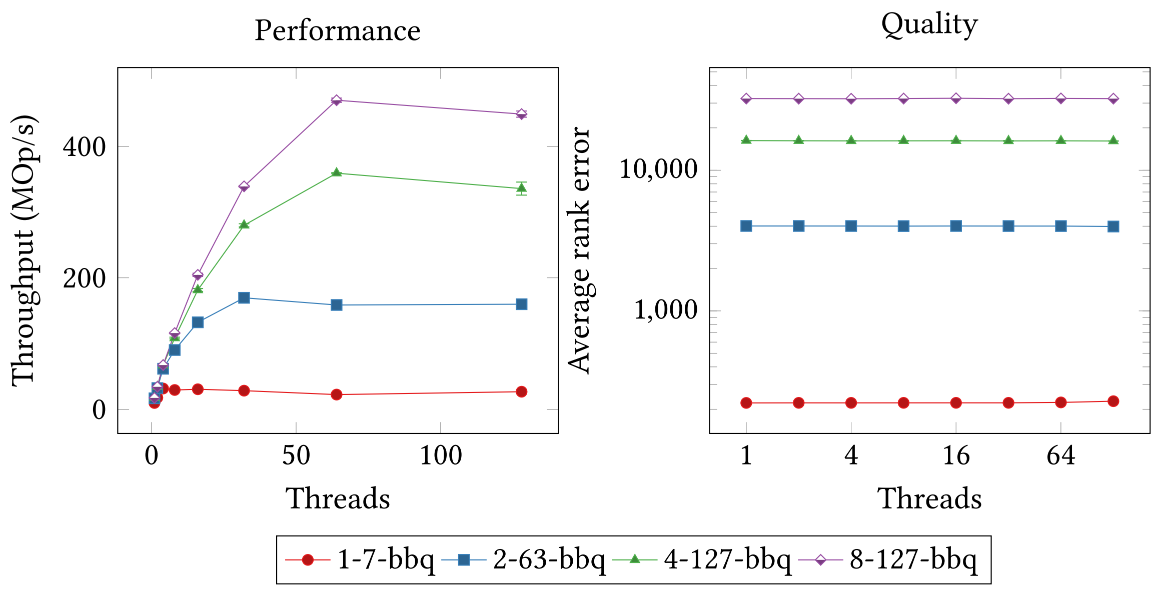
It might be interesting to further compare the quality of the queues with different producer-consumer ratios.

5.4.4. Strong Scaling

We want to make sure that our observed increases in performance of the BBQ cannot solely be attributed to the increased level of relaxation with increasing thread counts.

We conduct the benchmark in 5.4.1, making fixed the relaxation parameters of the highest thread count for each queue.

Figure 5.11.: Strong scaling benchmark comparing performance and quality



The results in Figure 5.11 show that more relaxed variants do scale strongly up to 32/64 threads while the quality remains constant, showing it is solely linked to the level of relaxation. The `1-7-bbq` barely scales, as is to be expected of its low level of relaxation since contention remains high.

5.4.5. Single Source Shortest Path BFS

This benchmark and its graph instances are adapted from Williams et al. [15].

As a macrobenchmark we calculate the shortest distance between a source node and all other nodes in a graph ($G = (V, E)$) via a parallel BFS that is implemented using a degenerate parallel version of Dijkstra's algorithm with no edge weights and using a FIFO instead of a priority queue. We explicitly have to utilize the ability of Dijkstra's algorithm to handle the distance to a node being decreased, which for us means refusing to expand nodes to which we have discovered a more optimal path since they were pushed. This is because the relaxation causes this exact situation to occur while it cannot occur in a sequential BFS. The goal is determining whether this potential redundant work is able to be compensated for by introducing parallelism.

We benchmark with two types of graphs, road networks of divisions of the USA[4] as well as one of Germany[5] and random hyperbolic graphs (rhg) that were generated by a modified version of the KaGen framework [3] where rhg$x$ has 2^x nodes.

Before examining the results it must be noted that our `1-7-bbq` failed to correctly complete the road network based benchmarks for $p < 64$ due to unknown reasons. We assure correctness by comparing the longest path within the graph among benchmark execution as well as counting the number of pushed and handled nodes. Due to time constraints it will not be possible to investigate this issue further for the time being.

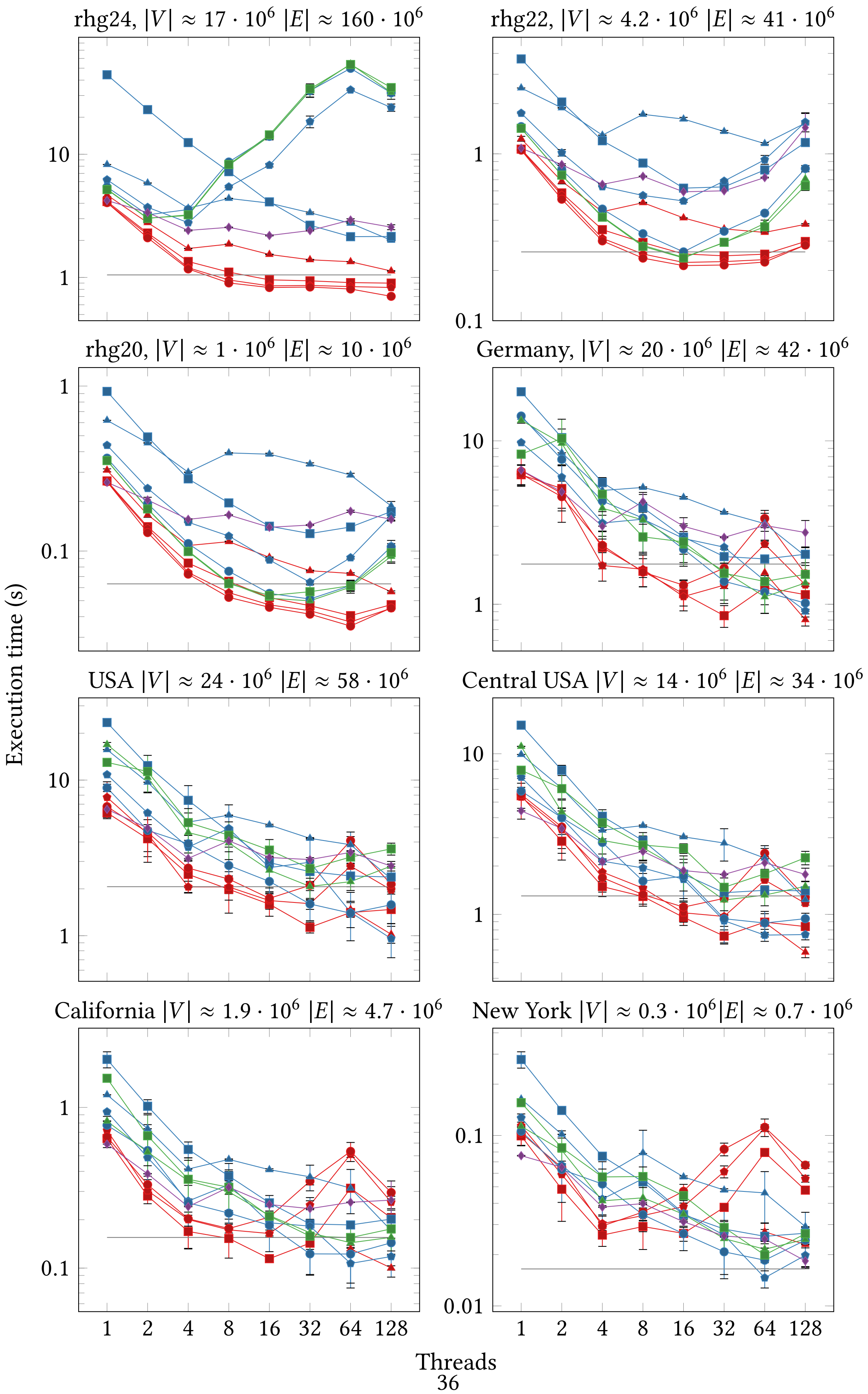
As can be seen in Figure 5.12, the sequential baseline algorithm puts all other results into perspective, showing that parallelizing here while achieving positive performance gain is genuinely nontrivial. Still, we are able to beat it in all but two instances, more or less closely depending on the instance. In general it appears that both rhg and small road network graphs are hard to beat the sequential approach in, suggesting that parallelism is most appropriate for large road networks graphs and similar.

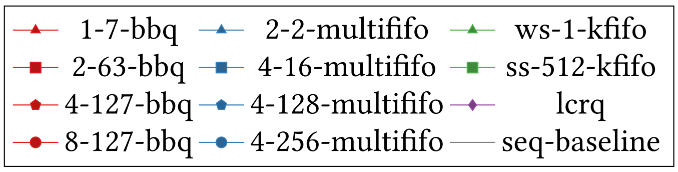
In comparison to the other concurrent queues our BBQ dominates the rhg benchmarks, while it struggles to compete with any of the more relaxed variations in the road network based ones. They still show good scaling behavior up until $p = 32$ at which point their performance plummets, slightly recovering at $p = 128$. Especially for the New York graph, which is the smallest, the BBQ heavily loses out to all other implementations, suggesting that it benefits from large graph sizes and struggles with particularly small graphs. The `1-7-bbq` is still able to be fairly competitive in the road network based benchmarks, only being beaten out slightly by the `4-128-multififo` in the California graph as well as by both the `4-128-multififo` and the LCRQ in the New York graph. A non-relaxed queue being the best in the smallest graph generally suggests relaxation only becoming beneficial for graphs of sufficient size.

[4]`https://www.diag.uniroma1.it/challenge9/download.shtml`

[5]`https://i11www.iti.kit.edu/resources/roadgraphs.php`

Figure 5.12.: Comparing SSSP performance on different graphs via a parallel BFS in log-log scale





The MultiFIFO performs quite poorly in the rhg benchmarks, but very competitively in the road network based ones. An interesting observation is that its best performing configuration for all road network based graphs is the second-most relaxed one, while for our BBQ it was the least-relaxed configuration that performed the best there.

The k-FIFO performs a lot better than what would have been expected based on the microbenchmarks and it especially scales quite well at higher core counts in the road network based graphs, managing to beat out both some BBQ and MultiFIFO configurations.

In general, while size is most definitely a factor relating to the differences in performance of implementations, it does not appear to be the deciding one, nor sufficient in explaining the stark contrast between road network and rhg instances. Thus, we observe a different metric of the graphs that appears to clearly separate the road networks from the rhg – average node degree, as seen in Figure 5.13.

Drawing on our results from 5.4.3, there appears to exist a correlation between the average node degree and the performance difference between producer-heavy and consumer-heavy ratios in the producer-consumer benchmark. It appears that configurations that perform well with a small amount of producers (thus having push performance superior to pop performance) like the BBQ tend to perform better on graph instances with a high node degree and vice-versa. This may be explained by every node on average having more connected nodes to be pushed successively by a thread expanding the node, while the remaining threads may consume the many results of this single node expansion. In the case of a low average node degree every thread must shortly after having expanded a node pop another one to expand.

Figure 5.13.: Average node degrees of different graph instances

Graph instance	Avg. node degree $\frac{2\|E\|}{\|V\|}$
rhg24	18.8
rhg22	19.5
rhg20	20.0
GER	4.2
USA	4.8
CTR	4.9
CAL	4.9
NY	4.7

Another noteworthy observation is that the deviation of measurements is a lot more significant on road network instances, which suggests the performance of rhg instances does not depend on the non-deterministic order in which nodes are explored, while the order of node exploration is a lot more relevant in road networks. This might be explained by the uniform nature of rhgs, whereas road networks tend to be chaotic. This begs the question of whether rhgs even benefit from FIFO semantics in a major way, which is why it may be interesting to further observe and compare performance with a concurrent bag, which should offer much higher raw performance while offering no FIFO semantics. Further supporting this hypothesis is the significantly higher performance of the `1-7-bbq` on road network instances, which may be attributed to its significantly higher quality.

6. Conclusion

We designed a concurrent FIFO queue that scales very well with an increasing number of processors and especially well until a very high degree of relaxation, where prior approaches plateau much quicker. It outperforms all competing approaches in all micro and all but three instances of the macro benchmark by being configurable to a wide array of different compromises between performance and quality. There exists no "one-size-fits-all" configuration although judging by the different performance characteristics of competing implementations it is doubtful whether one such configuration exists and achieving optimal performance in different benchmarks with different configurations while relying on the same principal approach appears sufficient.

6.1. Future Work

There exist various avenues of differing complexity and scale that could be explored to improve upon the BBQ further. The ones already conceived will be listed here in order of increasing complexity.

1. Writers autonomously abandoning their block if they have fallen too far behind the write window to increase quality and allow for giving strong quality guarantees by avoiding new elements being inserted far back into the queue. However, this would likely force push operations to check global shared state on each execution, which might negatively affect performance.
2. Forcing writers to claim a new block in the updated write window after a block was first claimed by a reader might decrease contention. This can be achieved by modifying the epoch on the first read claim.
3. Creating an unbounded variant by having windows as nodes and creating new nodes on demand (when the queue is full) seems relatively trivial while likely retaining much of the performance as the bulk of operations happens within the windows so the beneficial cache effects remain.
4. Allowing multiple writers within one block might allow for improving performance, quality and space utilization in some cases, as well as enabling the implementation of a linearizable fullness check.

5. Similarly to the LCRQ we might be able to further improve performance by replacing some of our inner-block CAS operations with FAA, *if* CAS retries can be shown to have a significant impact on performance.

6. The contention of all readers on the few remaining blocks within a window is likely a major bottleneck. One approach that could be taken here is allowing readers to be spread over multiple (two might be enough) windows, the major issue with this approach that would need to be solved is blocks potentially being left behind by readers becoming inactive.

7. Allowing the read window to be on top of the write window should both simplify the implementation, removing the need for forced window moving and its intricacies as well as benefiting performance, particularly within scenarios where there are few elements in the queue, as it prevents the read and write windows from "running around" the ring buffer. Although it might appear that this may result in infinitely bad quality if both windows continuously stay on top of each other and a block is "forgotten" by random chance, this is not the case, as each fully emptied block will still be sealed from use within the current epoch, which will eventually lead to all blocks being emptied and the write window advancing. The primary issue that would need to be addressed with this change is how to deal with the bitset, as it cannot be made entirely consistent for both window types (either readers find empty blocks marked as filled or writers find empty blocks they cannot write to). This issue would not occur when replacing the bitset as proposed in 6.1.2.

8. A fundamentally different approach to implementing the data structure which would also alleviate the contention described in item 6 could be realized by relaxing the fixed nature of the windows. Instead of dividing the circular buffer into windows which are further subdivided into blocks, immediately divide the circular buffer into blocks and interpret the windows as "sliding windows" encompassing a (potentially variable) amount of blocks which can be moved as little as one block at a time. This would result in potentially better performance and quality characteristics at the cost of an increased complexity in implementation.

9. In order to be able to properly compare our design with others, the other designs would need to be implemented in a fully optimized manner which might allow for drawing additional conclusions that may lead to improving the BBQ further.

Additionally outlined below are two fundamentally different approaches to accelerating the data structure potentially superior to using a single bitset, as this aspect is a major performance factor. The contention on the bitset is high, so finding ways to better spread this contention is likely to be beneficial to performance. Additionally, something we fail to utilize with higher block counts is that the blocks themselves adhere to strict FIFO ordering, which might be better utilized for increasing quality as well.

6.1.1. Hierarchical Bitsets

We find ourselves with the same problem as in the start: We have a singular point of contention, so why not simply spread it? By introducing a hierarchy of bitsets we would both achieve a reduction of contention on the one bitset, as well as benefiting from better quality if we utilize the additional bitsets well. The general idea would be to use the lowest level bitset for the first block claimed in the bitset and trying to spread later block claims among increasing levels. One simple approach to this would be keeping a thread-local counter which is reset on window update and using that to select a bitset so that the i-th block claimed by a thread would be marked in the bitset on level i. Finding a good solution here would likely be one of the primary challenges with realizing this approach. The amount of bitsets may be varied as well, realistically lying between B and $B \cdot p$ (the total amount of blocks per window), although approaching the upper bound would effectively mean having one bitset per block, which is unlikely to be ideal. There would also need to be a solution found for preventing duplicate claims on different levels.

6.1.2. Putting more FIFO in the FIFO

As mentioned in 4.1 our blocks really are just mini-FIFOs. We could apply the same concept to our windows. Instead of claiming blocks in random order, we could do so sequentially by essentially replacing our bitset with a block header and treating the blocks as cells. The important distinction would be that the read finished index may only be incremented by the thread that finishes reading the first still-filled block and it must further increase the index until it is on a block still containing elements. This means there will generally be four (potentially empty) contiguous segments within a window: read blocks, blocks that may or may not still contains elements but that did contain active readers at one point (touched blocks), blocks that are filled but have not been touched by a reader (unread blocks) and blocks that have not (yet) been filled (unfilled blocks). Finding a good metric for deciding between claiming an untouched block (decreasing quality) and a touched block (decreasing performance due to contention) is likely the primary challenge with this approach. Additional counters might help in making this decision (how many blocks in the touched segment are still unfinished, amount of readers on a block, etc). One benefit of this approach would be strict FIFO semantics for $p = 1$ and generally a better handle to control quality with via the aforementioned metric.

Bibliography

[1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. “Quasi-linearizability: Relaxed consistency for improved concurrency”. In: *Principles of Distributed Systems: 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings 14*. Springer. 2010, pp. 395–410.

[2] Faith Ellen, Danny Hendler, and Nir Shavit. “On the inherent sequentiality of concurrent objects”. In: *SIAM Journal on Computing* 41.3 (2012), pp. 519–536.

[3] Daniel Funke et al. “Communication-free massively distributed graph generation”. In: *Journal of Parallel and Distributed Computing* 131 (2019), pp. 200–217.

[4] Andreas Haas et al. “Scal: A benchmarking suite for concurrent data structures”. In: *Networked Systems: Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers 3*. Springer. 2015, pp. 1–14.

[5] Maurice P Herlihy and Jeannette M Wing. “Linearizability: A correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[6] Moshe Hoffman, Ori Shalev, and Nir Shavit. “The baskets queue”. In: *Principles of Distributed Systems: 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings 11*. Springer. 2007, pp. 401–414.

[7] ISO. *ISO/IEC 14882:2024 Information technology — Programming languages — C++*. Seven. Geneva, Switzerland: International Organization for Standardization, Oct. 2024, p. 2104. URL: `https://www.iso.org/standard/83626.html`.

[8] Christoph M Kirsch, Michael Lippautz, and Hannes Payer. “Fast and scalable, lock-free k-FIFO queues”. In: *Parallel Computing Technologies: 12th International Conference, PaCT 2013, St. Petersburg, Russia, September 30-October 4, 2013. Proceedings 12*. Springer. 2013, pp. 208–223.

[9] Nir Shavit Maurice Herlihy. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2012.

[10] Maged M Michael and Michael L Scott. “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”. In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 1996, pp. 267–275.

[11] Adam Morrison and Yehuda Afek. “Fast concurrent queues for x86 processors”. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 103–112.

[12] Christos H Papadimitriou. “The serializability of concurrent database updates”. In: *Journal of the ACM (JACM)* 26.4 (1979), pp. 631–653.

[13] Raed Romanov and Nikita Koval. “The state-of-the-art LCRQ concurrent queue algorithm does NOT require CAS2”. In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 14–26.

[14] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. “A practical nonblocking queue algorithm using compare-and-swap”. In: *Proceedings Seventh International Conference on Parallel and Distributed Systems (Cat. No. PR00568)*. IEEE. 2000, pp. 470–475.

[15] Marvin Williams, Peter Sanders, and Roman Dementiev. “Engineering multiqueues: Fast relaxed concurrent priority queues”. In: *arXiv preprint arXiv:2107.01350* (2021).

A. Appendix

Figure A.1.: Comparing a strong scaling k-FIFO with $k = 512$ with the weak scaling equivalent of $k = 4$ via the alternating push pop benchmark

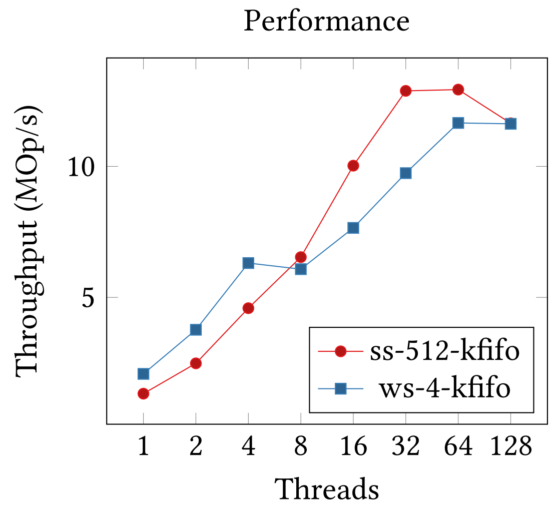


Figure A.2.: Comparing LPRQ and LCRQ via the alternating push pop benchmark

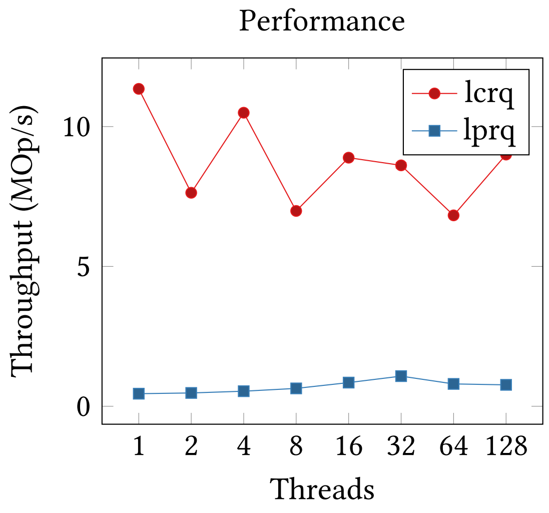


Figure A.3.: Comparing different BBQ configurations on a 80-core Neoverse-N1 ARM64 machine

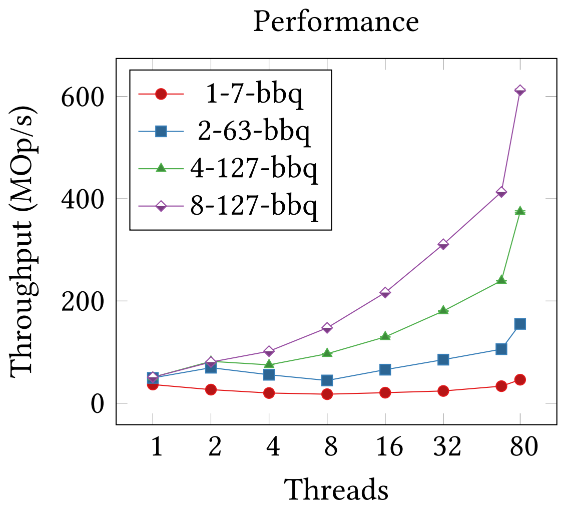
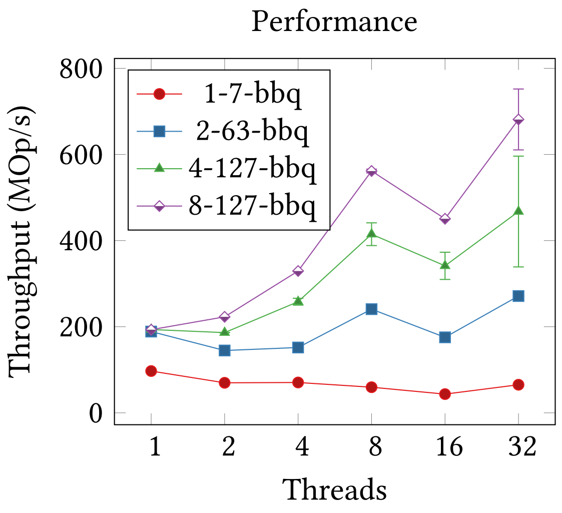


Figure A.4.: Comparing different BBQ configurations on a 16-core AMD Ryzen 9 7950X on Windows 10 compiled using MSVC 19.41.33923


Performance
Throughput (MOp/s)
Threads
1-7-bbq
2-63-bbq
4-127-bbq
8-127-bbq