# BlockFIFO & MultiFIFO: Scalable Relaxed Queues

Stefan Koch* Peter Sanders* Marvin Williams*

October 16, 2025

**Abstract.** FIFO queues are a fundamental data structure used in a wide range of applications. Concurrent FIFO queues allow multiple execution threads to access the queue simultaneously. Maintaining strict FIFO semantics in concurrent queues leads to low throughput due to high contention at the head and tail of the queue. By relaxing the FIFO semantics to allow some reordering of elements, it becomes possible to achieve much higher scalability. This work presents two orthogonal designs for relaxed concurrent FIFO queues, one derived from the MultiQueue and the other based on ring buffers. We evaluate both designs extensively on various micro-benchmarks and a breadth-first search application on large graphs. Both designs outperform state-of-the-art relaxed and strict FIFO queues, achieving higher throughput and better scalability.

**1 Introduction.** FIFO (First-in first-out) queues are data structures that support the insertion of elements (`push`) and the deletion of the least recently inserted element (`pop`). They are at the core of a wide range of applications, including breadth-first search, processing pipelines, message queues, and network routers. Sequential implementations using circular arrays are fast and cache-efficient, offering high throughput with constant-time operations. As most performance-critical computations today use parallel hardware, concurrent queues have become increasingly important. Indeed, in many applications, a concurrent queue serves as the central coordination mechanism among threads. For example, multiple threads may concurrently produce and consume data items from the queue, or, in a breadth-first traversal of a graph, the queue may hold the nodes yet to be explored.

Ideally, using $p$ threads, one would like the throughput to be close to $p$ times the throughput of a sequential queue. Unfortunately, high contention on the head and tail pointers of the queue is unavoidable [5, 2], and even sophisticated implementations of concurrent queues with strict semantics (e.g., [17, 18, 20]) suffer from poor scalability. The throughput remains constant or might even deteriorate with increasing numbers of threads. Thus, such designs are not suitable for situations that require a high throughput with many threads.

In order to achieve higher scalability, the idea of relaxing the semantics of queues and similar data structures in order to benefit from reduced contention has become an increasingly relevant topic of study [1, 15, 12, 24, 26]. We say that a popped element has *rank error* $r$ when it was pushed later than $r$ other elements that are still in the queue.

In this paper, we introduce the *MultiFIFO* and the *BlockFIFO*, two concurrent relaxed FIFO queues with particularly good scalability. The MultiFIFO (see section 4) is an adaptation of the MultiQueue [19, 27, 26], a state-of-the-art relaxed priority queue that builds upon multiple internal strict queues. Pushed elements go to a random queue, along with their insertion time stamp. Pops remove the least recent element from two randomly chosen queues. The MultiFIFO has expected constant time per operation and inherits a rank error linear in the number of threads from the MultiQueue. By making threads reuse the same internal queues for a fixed number of consecutive operations, performance can be further increased at the cost of higher rank errors.

The BlockFIFO (see section 5) is based on a ring buffer of small FIFO *blocks*, where threads operate on distinct blocks to reduce contention. It exhibits even higher throughput than the MultiFIFO in its most relaxed configurations.

*Summary of Contributions.*

- Introduction of two highly scalable relaxed concurrent FIFOs: the MultiFIFO and the BlockFIFO.
- Extensive experiments (section 6) on different architectures with various benchmarks indicating an order-of-magnitude improved throughput of the new FIFO queues compared to the previous state of the art.

**2 Preliminaries.** The first-in first-out (FIFO) queue is a data structure that manages a dynamic sequence of elements, supporting the following two operations:

*Karlsruhe Institut of Technology, Germany (stefan.koch@student.kit.edu, sanders@kit.edu, williams@kit.edu).

`push` Insert an element at the end (*tail*) of the sequence.

`pop` Remove and return the element at the front (*head*) of the sequence.

If a FIFO is empty, the `pop` operation *fails* and returns the special element $\perp$, which cannot be inserted. *Bounded* FIFO queues have a fixed capacity of elements, and pushing into to a full bounded FIFO conventionally fails[1]. The *ring buffer* (or *circular buffer*) is a common bounded FIFO queue, which conceptually arranges the slots for elements in a circular manner. Ring buffers are typically implemented as contiguous fixed-size arrays with head and tail pointers indicating the next slot to pop from and push into, respectively. These pointers are incremented by the corresponding operations and wrap around to the beginning when they reach the end of the array. *Concurrent* FIFO queues allow concurrent `push` and `pop` operations by multiple threads.

*Linearizability* [13] is a popular consistency criterion for specifying the semantics of concurrent data structures with respect to their sequential counterparts. Essentially, a concurrent data structure is linearizable if each operation appears to execute instantly at a single point in time (its *linearization point*) between its invocation and response. Linearizability is generally highly desirable: with non-overlapping operations, the data structure behaves like a sequential one, and concurrent executions offer strong and intuitive guarantees. Unfortunately, linearizable FIFO queue implementations suffer from high contention and exhibit limited scalability [5, 2]. To alleviate this issue, the semantics of FIFO queues can be *relaxed*, allowing the `pop` operation to remove elements out-of-order, or even *fail* to remove elements that are still in the queue. For many applications, it is useful for failed pops to be linearizable, meaning that popping may only fail if the queue was empty at some point during the operation. A natural quality metric for the degree of relaxation is the *rank error*: For an element $e$ that was returned by a `pop` operation, the rank error of the `pop` operation is the number of elements in the queue that were inserted before $e$.

An algorithm is *lock-free* if at least one thread is guaranteed to make progress towards completing an operation within a bounded number of steps, regardless of the actions of other threads.

The *ABA problem* arises when a thread reads the same value $A$ from a shared memory location twice, incorrectly assuming the value has not changed between the reads, even though it was changed to $B$ and then back to $A$ by other threads.

[1]Other semantics, such as overwriting the oldest element or allocating more space, are possible.

We denote $p$ as the (maximum) number of threads participating in an algorithm concurrently. An event occurs *with high probability* (with respect to $p$) if the probability is at least $1 - p^{-a}$ for some constant $a \geq 1$.

**3 Related Work.** The FIFO queue is a conceptually simple data structure that appears in most introductory algorithm textbooks (e.g., [4]) and is readily available in the standard library of most programming languages. As a core component in a wide range of applications, including message queues, processing pipelines, and breadth-first searches, FIFO queues have been studied extensively in the literature. Here, we focus on concurrent FIFO queues with an emphasis on relaxed variants.

**Linearizable FIFO Queues.** One of the earliest linearizable lock-free concurrent FIFO queue algorithms is the *MS-Queue*, proposed by Michael and Scott [17]. The algorithm is based on a linked list, where nodes can be appended and removed in a lock-free manner. To avoid the ABA problem during atomic updates of next-pointers, each pointer is tagged with a version number. The design suffers from high contention on the head and tail pointers, which limits its scalability. Nevertheless, its simplicity and ease of implementation have made it the foundation for numerous subsequent designs (e.g., [11, 7, 14]). One notable example is the *Baskets Queue* by Hoffman et al. [14], which uses unordered *baskets* for concurrently inserted elements as nodes in the linked list.

Tsigas and Zhang [23] propose a bounded, lock-free FIFO queue based on ring buffers. Although bounded queues are less flexible than unbounded ones, they offer several practical advantages: they do not require dynamic memory allocation, are generally more cache-friendly, and do not require complex memory reclamation mechanisms.

The LCRQ (List of Concurrent Ring Queues), introduced by Morrison and Afek [18], is a state-of-the-art concurrent FIFO queue design. It combines the MS-Queue with ring buffers to leverage the advantages of both data structures. Its implementation favors the more efficient atomic fetch-and-add operations over compare-and-swap operations. Both the MS-Queue and the LCRQ rely on the double-width compare-and-swap operation (dCAS), which most platforms do not support natively. The LPRQ (*portable* LCRQ) [20] improves the portability of the LCRQ by eliminating the dCAS while maintaining comparable performance.

**Relaxed FIFO Queues.** Afek et al. [1] introduce the *Segmented Queue*, a relaxed FIFO queue based on a linked list of *segments*, where each segment is a static array of size $C$. When pushing, a thread writes the

element into a random empty cell in the tail segment, appending a new segment if necessary; popping is performed analogously. The Segmented Queue exhibits bounded worst-case rank errors in $\mathcal{O}(C)$. The *k-FIFO* queue by Kirsch et al. [15] enhance this design with scalable, linearizable emptiness checks.

A popular approach to relaxing the semantics of a data structure is to employ multiple (thread-safe) instances of the data structure and distribute the access to them among the threads (e.g., [11, 21, 24]). The distribution mechanism is crucial for the performance and quality guarantees of these designs. Rukundo et al. [21] introduce the *2D-Queue*, which distributes operations among its internal queues such that their sizes remain within a fixed range. This design guarantees a bounded worst-case rank error of $\mathcal{O}(wr)$, where $w$ is the number of queue instances and $r$ is the length of the range. The $d$-RA ($d$-randomized load balancer) by Haas et al. [11] samples $d \geq 1$ data structure instances for each operation, selecting the least-loaded for pushes and the most-loaded for pops. However, von Geijer et al. [24] demonstrate that this balancing scheme can lead to increasing rank errors with growing queue sizes. They propose the *d-CBO* ($d$-Choice Balanced Operations), which balances using dedicated push and pop counters per queue instance. A `push` operation samples $d \geq 2$ instances and chooses the one with the fewest prior pushes. Symmetrically, a `pop` operation samples $d$ instances and chooses the one with the fewest prior pops. Interestingly, this scheme stabilizes the rank errors empirically. Concurrent bags [22] impose no ordering on the elements, thus can be viewed as the most extreme version of relaxed FIFO queues.

Henzinger et al. [12] introduce the *Quantitative Relaxation* framework to specify and analyze semantics of relaxed concurrent data structures. However, the framework is not applicable to randomized algorithms where the rank errors are not bounded.

**4 The MultiFIFO.** The MultiQueue [19, 27, 26] is a state-of-the-art relaxed concurrent priority queue.

It consists of an array of $c \cdot p$ sequential priority queues, each protected by an mutual exclusion lock, where $c \geq 2$ is the *queue factor*. The operations are handled asymmetrically: An insertion randomly selects and locks a single queue to insert the new element. A deletion uses the *power of two choices* principle by randomly sampling *two* queues, and locking the one containing the element with the highest priority to perform the deletion. If a thread fails to acquire a lock during either operation, it retries the operation.

Despite employing locks, the MultiQueue is *probabilistically wait-free*, meaning that all operations of all threads make progress within a bounded number of steps in expectation. To improve the performance of the MultiQueue, Williams et al. [27] propose the concept of *stickiness*, where threads reuse the same two queues for $s$ (the *stickiness period*) consecutive operations. For details on the MultiQueue, we refer to the original paper [27, 26].

A promising approach to creating a relaxed FIFO queue is to adapt the MultiQueue. For this adaptation, which we call the *MultiFIFO*, we replace the sequential priority queues with in-place ring buffers and tag each element with its insertion timestamp. The pop operation randomly samples two ring buffers, compares their head elements, and removes the one with the earlier timestamp.

Due to the similarity to the MultiQueue, the MultiFIFO inherits many of its desirable properties, such as *probabilistic wait-freedom* [26], linear (in $p$) expected rank errors of $\frac{5}{6}cp - 1 + \frac{1}{6cp}$ (see [25]), and rank errors in $\mathcal{O}(p \log p)$ with high probability. Both operations are in $\mathcal{O}(1)$ in expectation, since finding an unlocked ring buffer takes constant time in expectation (see [26]) and the time for the actual operations on the ring buffers is constant.

**5 The BlockFIFO.** In this section, we present the BlockFIFO, a bounded, lock-free, relaxed concurrent FIFO queue. We begin by outlining its basic design, then analyze its practical limitations and present several key improvements.

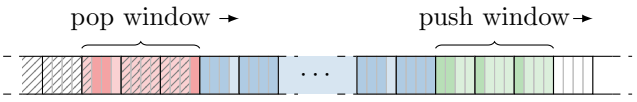


Figure 5.1: Schematic diagram of the BlockFIFO with block size $C = 4$ and window size $w = 3$, represented as a linear array. The colored area represents the "active" part of the array. Slots with stronger colors contain elements. The line pattern indicates that a slot is emptied and not used again.

**5.1 Basic Design.** Conceptually, the BlockFIFO is an infinite array of *blocks*, where each block is a small, fixed-size buffer of size $C$, which holds the actual elements. For the sake of simplicity, we present the data structure assuming an infinite, linear memory layout. We show a practical, lock-free implementation based on ring buffers in Appendix A.

A block can be in one of three states: *unclaimed*, *claimed*, or *closed*. An unclaimed block is empty and can be claimed for a `push` operation by any thread, after which it becomes claimed. No other thread may claim a claimed block or insert elements into it. Once all

**Algorithm 1:** Pseudocode for the `push` and `pop` operations. The `pop` operation returns the deleted element or $\perp$ if the queue was empty.

```
Function push(e)
    z ← currentPushWindow()
    k ← lastClaimedBlock()
    while ¬pushToBlock(z, k, e) do
        while ¬(k ← claimNewBlock(z)) do
            z ← advancePushWindow(z)

Function pop()
    z ← currentPopWindow()
    k ← lastPopBlock()
    while (e ← popFromBlock(k)) = ⊥ do
        repeat
            z ← advancePopWindow()
            if k ← findPopBlock(z) then
                break
            if isEmpty() then return ⊥
    return e
```

elements have been popped from a block, it is closed and can no longer be used. The *push window* and the *pop window* are the ranges of blocks that are currently available for `push` and `pop` operations, respectively. The *window size* $w$ (the number of blocks in each window) is linear in the number of threads $p$, resulting in $w = B \cdot p$ blocks, where $B \geq 1$ is the *block factor*. Initially, all blocks are unclaimed and the pop window is positioned directly behind the push window. Figure 5.1 shows the BlockFIFO with $C = 4$ and $w = 3$ schematically.

Algorithm 1 gives high-level pseudocode of the `push` and `pop` operations. When pushing an element, a thread first tries to reuse the last block it claimed. The method `pushToBlock(`$z, k, e$`)` appends the element $e$ to block $k$, failing if $z$ is not the current push window or block $k$ is full. If it fails, the thread attempts to claim a new block in the current push window (`claimNewBlock(`$z$`)`). To find an unclaimed block, it linearly scans the push window, starting from a random position. If there is no unclaimed block, the thread advances the push window by $w$ blocks and retries the scan. The method `advancePushWindow(`$z$`)` only advances the push window if it is currently $z$, otherwise it returns the current push window. Note that not all blocks have to be full for the push window to advance.

When popping, a thread first tries to pop from the same block it used for the last `pop` operation. If the block is not closed, the block might still contain elements and the pop window did not move past it. The method `popFromBlock(`$k$`)` removes the head element from block $k$, returning $\perp$ if the block was already empty. If the block is empty afterwards, it closes the block.

To find a new block to pop from, the thread first advances the pop window (`advancePopWindow()`) past any leading closed blocks, while ensuring the window does not overlap the push window. The thread then scans the current pop window linearly for a non-closed block, starting from a random block in the window $z$ (`findPopBlock(`$z$`)`). In case the pop window is empty and cannot advance, the thread checks whether the push window contains any elements. If it does, it advances both windows by the full window width and retries the operation. If the push window is also empty, the queue is determined to be empty and the `pop` fails (`IsEmpty()`). Unlike pushes, different threads must be allowed to pop from the same block.

### 5.2 Theoretical Properties.

*Failed pops are linearizable.* For a `pop` operation to fail, the pop window must be positioned directly behind the push window, all blocks in the pop window must be closed, and the push window must be empty. After checking these conditions, the thread verifies that the push window did not move in the meantime. Since no new elements can be inserted into closed blocks and no elements can be deleted from the push window, the queue must have been empty just after scanning the pop window and finding only closed blocks.

*Asymptotic Considerations.* We do not have a complete analysis of the BlockFIFO yet. However, the design is based on two principles: By using windows with $\Omega(p)$ blocks, threads can mostly work on "their" local insertion and deletion buffer blocks. By using blocks of size $\Omega(p)$, expensive operations like moving windows and searching for blocks can be amortized over $\Omega(p)$ fast, local, cache-efficient operations on local blocks. This implies constant time operations at the price of rank errors of size $\Omega(p^2)$; we view it as likely that this is also an upper bound.

This reasoning breaks down when the queue is almost empty. In that case expensive global operations dominate. The BlockFIFO is designed for situations where the queue is sufficiently full ($\Omega(p^2)$ elements) most of the time.

**5.3 Practical Improvements.** Searching for a new block to operate on can be expensive, especially when only few suitable blocks are available. We therefore augment the data structure with a bitset, where a bit indicates whether the corresponding block has been closed. When a thread claims a new block for pushing, it

sets the corresponding bit to one. When a thread empties a block by popping from it, it sets the corresponding bit to zero. The bitset improves the cache locality for the block search, since the individual blocks do not need to be inspected. Further, multiple bits can be inspected simultaneously with SIMD (single instruction, multiple data) techniques. Appendix B gives details about the implementation of the bitset.

Another performance bottleneck in our design is the contention when multiple threads pop elements from the same block. To alleviate this, we employ a *lookahead* window of $w$ blocks in front of the current pop window. A popping thread first searches for *fresh* blocks in the pop window, i.e., blocks that have not yet been popped from by other threads, analogously to the `push` operation. If no fresh block is available, it searches for one in the lookahead window, before considering non-closed blocks in the push window. Since the pop window can only advance when the first block in it is closed, another promising approach is to bias the random block selection towards the front of the pop window. While this may increase the contention on the first blocks, it facilitates faster advancement of the pop window to make new blocks available faster. We did not investigate this approach beyond preliminary experiments.

## 6 Evaluation.

We evaluate and compare the BlockFIFO and MultiFIFO with state-of-the-art FIFO queues across multiple workloads and different hardware architectures. This includes micro-benchmarks and a concurrent breadth-first search on various graphs.

### 6.1 Methodology.

To measure the maximum throughput, we design the following two micro-benchmarks that aim to mimic practical workloads.

- **Push-Pop**: Each thread repeatedly performs alternating `push` and `pop` operations for 5 seconds, keeping the number of elements in the queue constant. We choose a large enough capacity for the bounded queues and pre-fill all queues with sufficiently many elements to ensure that the queues never run full or empty. We measure throughput as the number of `push`–`pop` iterations per second.
- **Producer-Consumer**: Given a fixed number of threads, some perform `push` operations, while the other perform `pop` operations. We use the same queue size and capacity as in the push-pop benchmark. Depending on the configuration, the number of elements may grow or shrink, in extreme cases the queues may even run full or empty. We measure the throughput as the minimum of `push` and `pop` operations performed per second.

The elements in the queue are 64-bit integers, since this data type is supported by all implementations.[2] Each experiment is repeated five times, and we report the mean. The standard deviation is shown in plots as errors bars where significant.

To measure the rank errors exhibited by a queue, we take timestamps for each insertion and deletion. After the benchmark, we reconstruct a global order of operations, which we replay sequentially. While this process is not perfectly accurate, we deem it sufficient for our purposes.

The BlockFIFO, the MultiFIFO and all experiments are implemented in standard C++20. Competitors are implemented in C and C++, some using non-standard extensions. All code is compiled with GCC 14.2.0, using the flags `-O3 -DNDEBUG`. We pin execution threads to hardware threads to increase stability and consistency across experiments. We use three different machines (`AMD`, `ARM`, and `Intel`) for our benchmarks. Details to the hardware of these machines can be found in Table 6.1. All machines run Rocky Linux 9.5 with Linux kernel version 5.14. Unless specified otherwise, we use machine `AMD`. The source code for our implementation, including the data structures and the benchmarks, is available online[4].

### 6.2 Competitors.

We compare the BlockFIFO and MultiFIFO with the following implementations of state-of-the-art competitors found in the literature. While the main focus is on relaxed FIFO queues, we also include strict FIFO queues for reference. Relaxed FIFO queues are expected to outperform strict FIFO queues in terms of throughput and scalability at the cost of rank errors.

**BF** The BlockFIFO with configurable block factor $B$ and block size $C$.

**MF** The MultiFIFO with configurable queue factor $c$ and stickiness period $s$.

**k-FIFO** Implementation[5] of the $k$-FIFO [15, 10] with configurable segment size $k$. The authors suggest a segment size of $k = p$.

**d-CBO** Implementation[6] of the $d$-CBO [24] with configurable number of sub-queues per thread $c$.

**LCRQ** Implementation[7] of the LCRQ [18].

[2] The $k$-FIFO implementation reserves some higher-order bits for internal tagging, which are unavailable for data. This has no practical implications on our experiments.

[4] https://zenodo.org/records/17293832

[5] https://github.com/cksystemsgroup/scal

[6] https://github.com/dcs-chalmers/semantic-relaxation-dcbo

[7] https://zenodo.org/records/7337237

Table 6.1: Hardware details of all machines used in experiments.

| | CPU (ISA) | Sockets/Cores/Threads | Max. Clock Freq. | L1d/L2/L3 Cache[3] |
|---|---|---|---|---|
| `AMD` | EPYC 9684X (x86-64) | 1/96/192 | 3.7 GHz | 32 KiB/1 MiB/1152 MiB |
| `ARM` | Neoverse-N1 (ARMv8.2-A) | 1/80/80 | 3.0 GHz | 64 KiB/1 MiB/- |
| `Intel` | Xeon Gold 6138 (x86-64) | 4/80/160 | 3.7 GHz | 32 KiB/1 MiB/27.5 MiB |

**FAAAQueue** Implementation[8] of the FAAAQueue.

**6.3 Micro-Benchmarks.** Figure 6.1 shows the throughput and quality of all competitors for a wide range of parameter configurations on the push-pop benchmark. The BlockFIFO and the MultiFIFO are the only competitors where the throughput increases significantly with higher degrees of relaxation. Consequently, they achieve an order of magnitude higher throughput than the next fastest competitor (the $d$-CBO) at the cost of higher rank errors. With lower thread counts ($p = 4$, $p = 32$), the BlockFIFO dominates the MultiFIFO with higher throughput at similar rank errors. However, with $p = 192$ threads, the MultiFIFO achieves higher throughput than the BlockFIFO for rank errors below 2000. With more relaxed configurations, the throughput of the BlockFIFO scales faster, outperforming the MultiFIFO significantly. The $d$-CBO offers slightly lower rank errors than the highest-quality configurations of the MultiFIFO and BlockFIFO while achieving similar throughput. Unfortunately, the throughput of the $d$-CBO barely increases with higher degrees of relaxation. The $k$-FIFO is dominated by other competitors on the entire Pareto-front. While it exhibits similar quality to the $d$-CBO, it has significantly lower throughput and also does not scale well with higher degrees of relaxation. As expected, the strict FIFO queues exhibit virtually no rank errors. Their throughput is competitive with the highest-quality configurations of the other competitors for low thread counts, but is an order of magnitude slower for 192 threads.

For subsequent experiments, we select three configurations (**Q**uality, **B**alanced, and **F**ast) of implementations with tunable quality parameters that are Pareto-optimal on the throughput-quality spectrum for the push-pop benchmark with 192 threads on machine `AMD`. The selected configurations are given in Table 6.2 and additionally annotated in Figure 6.1. Detailed parameter tuning for the BlockFIFO and the MultiFIFO can be found in Appendix C.

Figure 6.2 shows the throughput scaling behavior of all competitors on the push-pop benchmark for all machines. Unfortunately, the LCRQ and $d$-CBO implementations are not compatible with machine `ARM`, and are therefore omitted. The balanced and fast configurations of the BlockFIFO and MultiFIFO consistently achieving higher throughput than the other competitors for all thread counts, with the fast BlockFIFO being the fastest. While the $d$-CBO also exhibits some scalability, its fast configuration is an order of magnitude slower than the fast BlockFIFO and MultiFIFO configurations. The quality variants show similar performance to the $d$-CBO with the highest thread count. The quality configuration of the MultiQueue and the $d$-CBO behave almost identically due to their conceptual similarity. The $k$-FIFO does not scale well, and—unsurprisingly—the strict queues do not scale at all. The results are generally consistent across all machines, with the notable exception of the BlockFIFO on the Intel machine. Here, the quality and balanced configurations do not scale well when using more than one NUMA node ($p > 16$). This is probably due to the fact that all threads must access blocks within the current windows. These blocks likely reside on the same NUMA node, leading to high bus contention. The MultiFIFO does not suffer from this problem, since different threads can operate on different sub-queues that may be located on different NUMA nodes. Appendix E shows the quality scaling behaviour of all competitors on the push-pop benchmark for machine `AMD`.

Figure 6.3 shows the producer-consumer benchmark for different ratios of producers and consumers and different thread counts. Almost all competitors favor balanced ratios of producers and consumers over extreme imbalances, indicating that the performance of both operations is similar. Similar to the push-pop bench-

Table 6.2: Selected configurations for configurable competitors used in the experiments.

| Competitor (Parameters) | Q | B | F |
|---|---|---|---|
| BF $(B, C)$ | $1, 7$ | $1, 63$ | $1, 511$ |
| MF $(c, s)$ | $2, 1$ | $4, 16$ | $4, 256$ |
| $k$-FIFO $(k)$ | $p/2$ | $p$ | $4p$ |
| $d$-CBO $(c)$ | $p/2$ | $p$ | $2p$ |

[8] https://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html

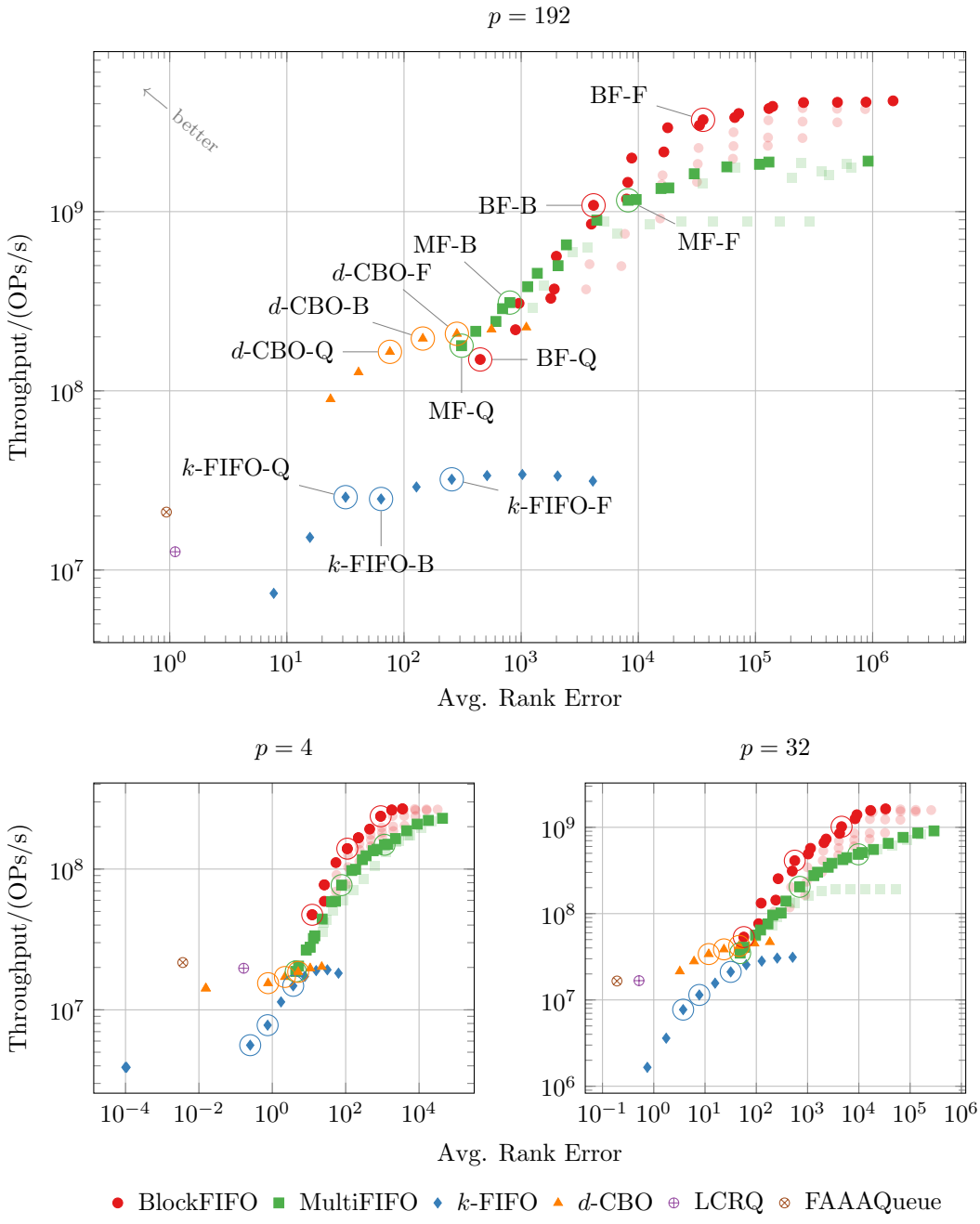


Figure 6.1: Various configurations of all competitors on the push-pop benchmark with different thread counts on machine `AMD`. For the BlockFIFO, the block factor $B$ ranges from 1 to 16 and the block size $C$ ranges from 7 to 2047. For the MultiFIFO, sub-queues per thread $c$ range from 2 to 8 and stickiness $s$ ranges from 1 to 4096. For the $k$-FIFO, segment size $k$ ranges from $\frac{1}{8}p$ to $64p$. For the $d$-CBO, the sub-queue count $c$ ranges from $\frac{1}{8}p$ to $8p$. For all ranges, we sample integer powers of two (minus one for the block size). Non-Pareto-optimal configurations of the BlockFIFO and MultiFIFO are shown with low opacity. Configurations used in further experiments are highlighted with circles and their name is annotated.

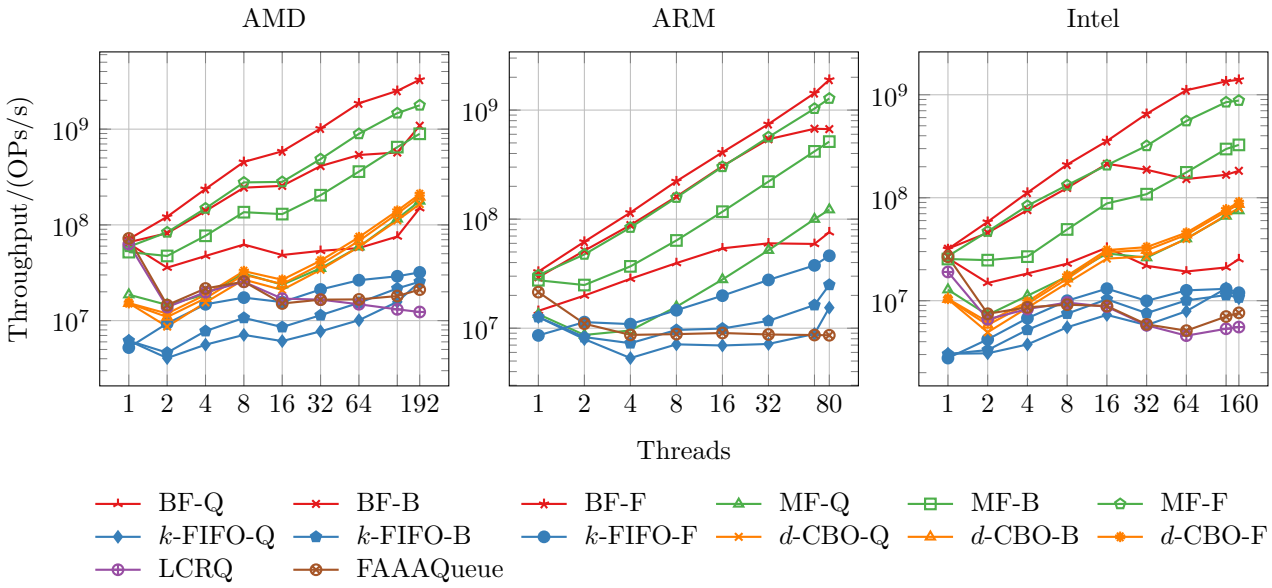


Figure 6.2: Throughput on the push-pop benchmark with different thread counts on all machines.

mark, the balanced and fast variants of the BlockFIFO and MultiFIFO generally outperform all other competitors. Interestingly, the BlockFIFO still performs well in consumer-heavy workloads, where the queue is likely to run empty, which is a scenario that it is not designed for. The MultiFIFO exhibits a bias towards producer-heavy workloads, with an optimal ratio of around $\frac{11}{16}$. However, it has the most pronounced performance drops for extreme ratios among all competitors. This is likely due to the fact that queues are locked by one type of worker for the majority of the time, while the other worker type starves, as there is no progress guarantee for each type of operation. The $d$-CBO is more stable across different ratios than the MultiFIFO for all thread counts because it avoids locking by using lock-free sub-queues. Both perform poorly with consumer-heavy workloads due to triggering a costly emptiness-detection algorithm that scans all sub-queues. The $k$-FIFO and the strict competitors exhibit very stable performance across all ratios, but they are consistently slower than the other relaxed competitors.

Results of the producer-consumer benchmark on the `ARM` and `Intel` machines are given in Appendix E.

**6.4 Breadth-First Search.** The single-source shortest path problem (SSSP) is a fundamental and well-known graph problem. On unweighted graphs, it can be solved with a Breadth-First Search (BFS). A straightforward BFS algorithm uses a FIFO queue to store the nodes that are to be explored. A natural parallelization of this algorithm is then to use a concurrent queue. Williams and Sanders [26] describe a parallel SSSP algorithm for weighted graphs that employs a concurrent priority queue. The idea of the algorithm is to allow the exploration of sub-optimal nodes, potentially requiring the re-exploration of parts of the graph when a shorter path to a node is found later. We adapt this algorithm to unweighted graphs by using a concurrent FIFO queue instead of a concurrent priority queue.

We evaluate the BFS on various real-world graphs for strong scaling behaviour and on random graphs (generated with KaGen [6]) for weak scaling behaviour. The real-world graphs are the road networks of Europe and the USA, the follower-relationships on Twitter [16], cross-references in the English Wikipedia and the network of *.uk* domains [3].[9] Table 6.3 compares core characteristics of these graphs. For the weak scaling experiments, we use the following graph classes:

- `RGG2D`: Random geometric graph with points on the 2D plane.
- `RHG`: Random hyperbolic graph with a gamma of $\gamma = 2.7$.
- `GNM`: Erdos-Rényi graph.

The number of nodes scales linearly with the number of threads with a scaling factor of $2^{16}$, resulting in $n = 2^{16}p$ nodes. The average vertex degree is set to 64.

[9]The road network graphs were obtained from https://i11www.iti.kit.edu/resources/roadgraphs.php, the others from https://law.di.unimi.it/datasets.php

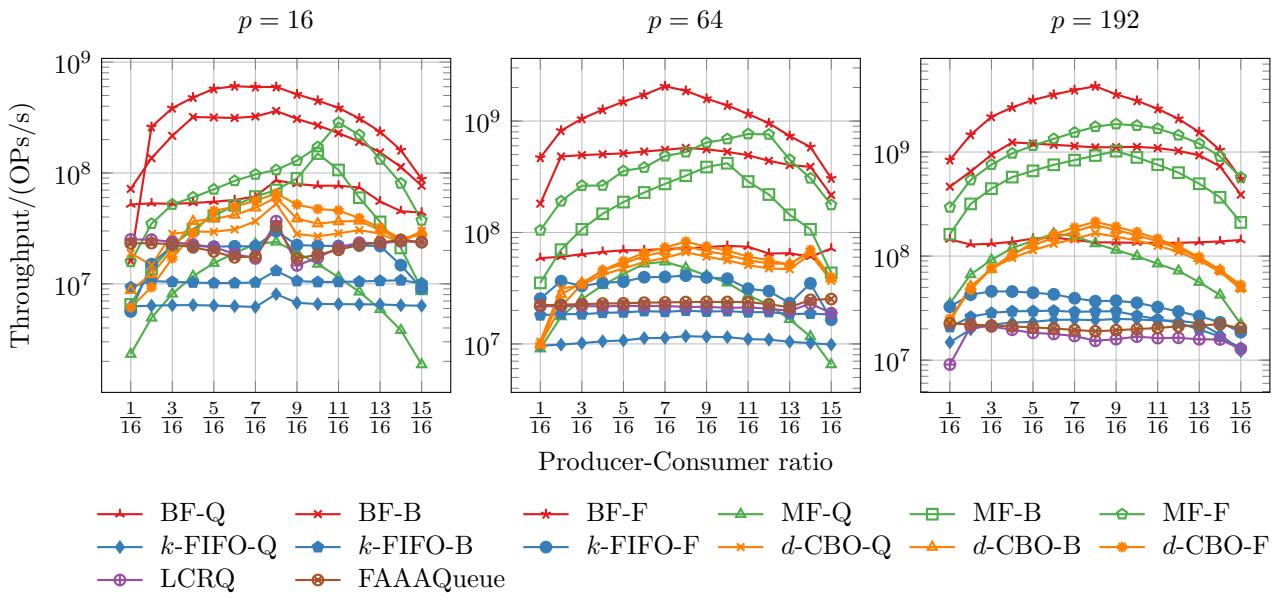


Figure 6.3: Throughput with different producer-consumer ratios at different thread counts.

Table 6.3: Number of nodes ($n$) and edges ($m$), as well as average node degrees for all graph instances used for strong scaling.

| Name | $n/10^6$ | $m/10^6$ | Avg. deg. |
|---|---|---|---|
| `enwiki-2022` | 6.5 | 159.0 | 49.0 |
| `ljournal-2008` | 5.4 | 79.0 | 29.5 |
| `twitter-2010` | 41.7 | 1468.4 | 70.5 |
| `uk-2005` | 39.5 | 936.4 | 47.5 |
| `OSM Europe` | 173.8 | 348.0 | 4.0 |
| `OSM USA` | 23.9 | 28.9 | 2.4 |

Figure 6.4 shows the performance on the strong scaling and weak scaling graphs. On the real-world graphs, the BlockFIFO and MultiFIFO generally scale well and achieve a speedup of up to 10 over the sequential BFS, except for the `OSM USA` graph. The BlockFIFO is consistently the best-performing queue for all thread counts and graph instances. On the `enwiki-2022`, `ljournal-2008` and `twitter` graphs, the $d$-CBO also has good scalability and competitive performance to the MultiFIFO. Particularly on the road networks, no competitor besides the BlockFIFO and the MultiFIFO is able to substantially outperform the sequential BFS even with 192 threads. The non-relaxed competitors do not scale beyond 8 threads and are consistently slower than all relaxed competitors.

The weak scaling graphs show similar results, except for the `RGG2D` graph class, where the $d$-CBO outperforms all other competitors for high thread counts. On all other graphs, the BlockFIFO and the MultiFIFO outperform the other competitors consistently. On the `RHG` graph, the other competitors are slower than the sequential algorithm except for the $d$-CBO with the highest number of threads.

Figure 6.5 shows the extra work incurred by the relaxed queues as the number of processed nodes divided by the number of processed nodes in a sequential BFS. Generally, the extra work is within one order of magnitude of the sequential BFS. Since the throughput of the relaxed queues is up to two orders of magnitude higher than strict queues, this is a worthwhile trade-off. While the MultiFIFO often induces the most extra work, the BlockFIFO is often competitive with the strict `FAAAQueue`. The `RGG2D` graph is a notable outlier, where both the BlockFIFO and MultiFIFO induce significantly more extra work than the other competitors. This aligns with the $d$-CBO outperforming them on this graph, but explaining the discrepancy requires further analysis.

In Appendix D, we show additional benchmarks for a BFS benchmark with multiple source nodes, resulting in significantly higher scalability. The BlockFIFO, the MultiFIFO, and the $d$-CBO achieve a speedup of over two orders of magnitude over the sequential execution.

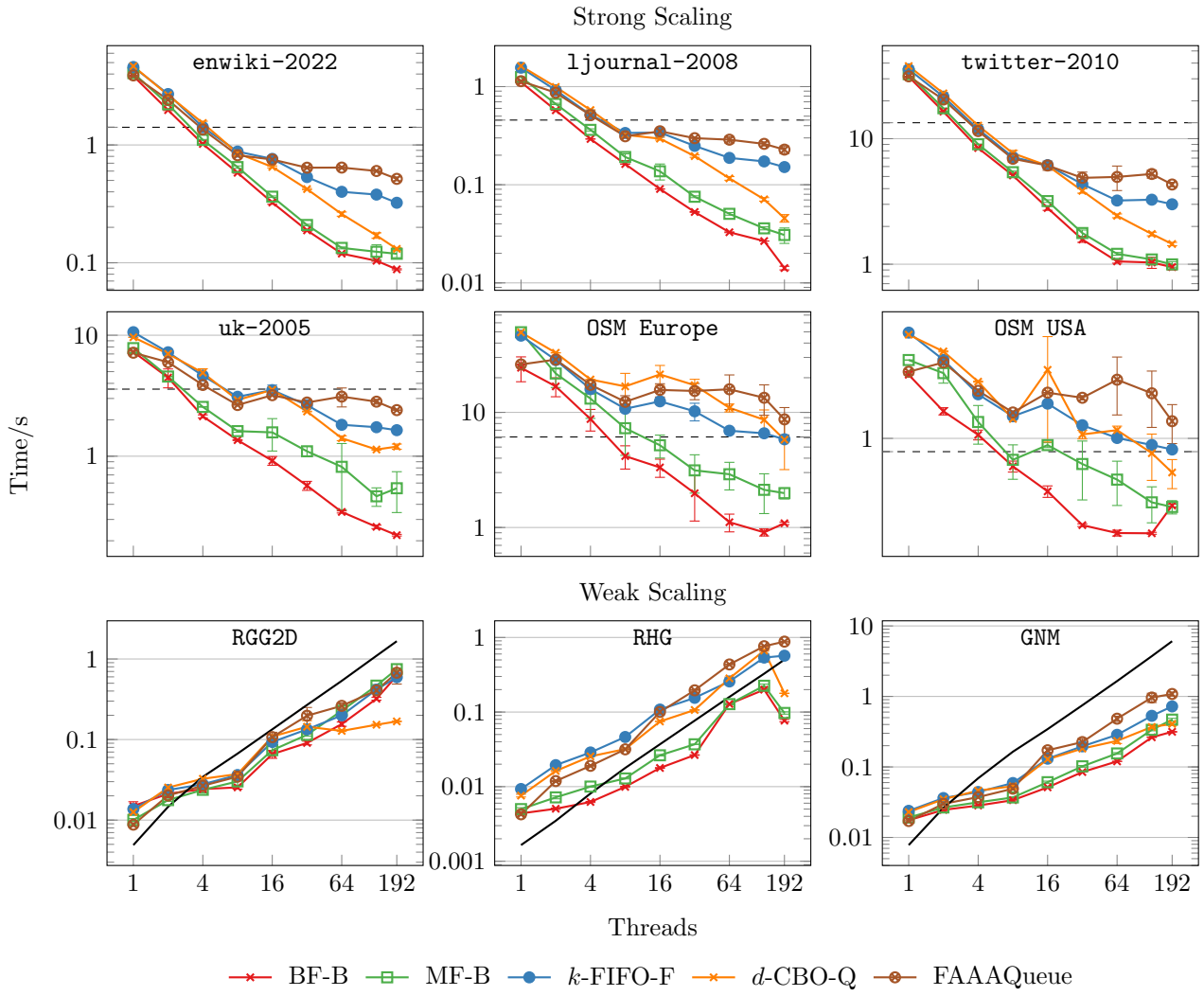


Figure 6.4: Weak and strong scaling BFS benchmarks. Only the best-performing queue configurations for each competitor are shown for clarity. The upper part shows strong scaling behaviour on real-world graphs, where the dotted lines represent the execution time of a sequential BFS. The bottom part shows weak scaling behaviour, where the graph size scales with the number of threads. The black line represents the time required by a sequential BFS on the scaled graph.

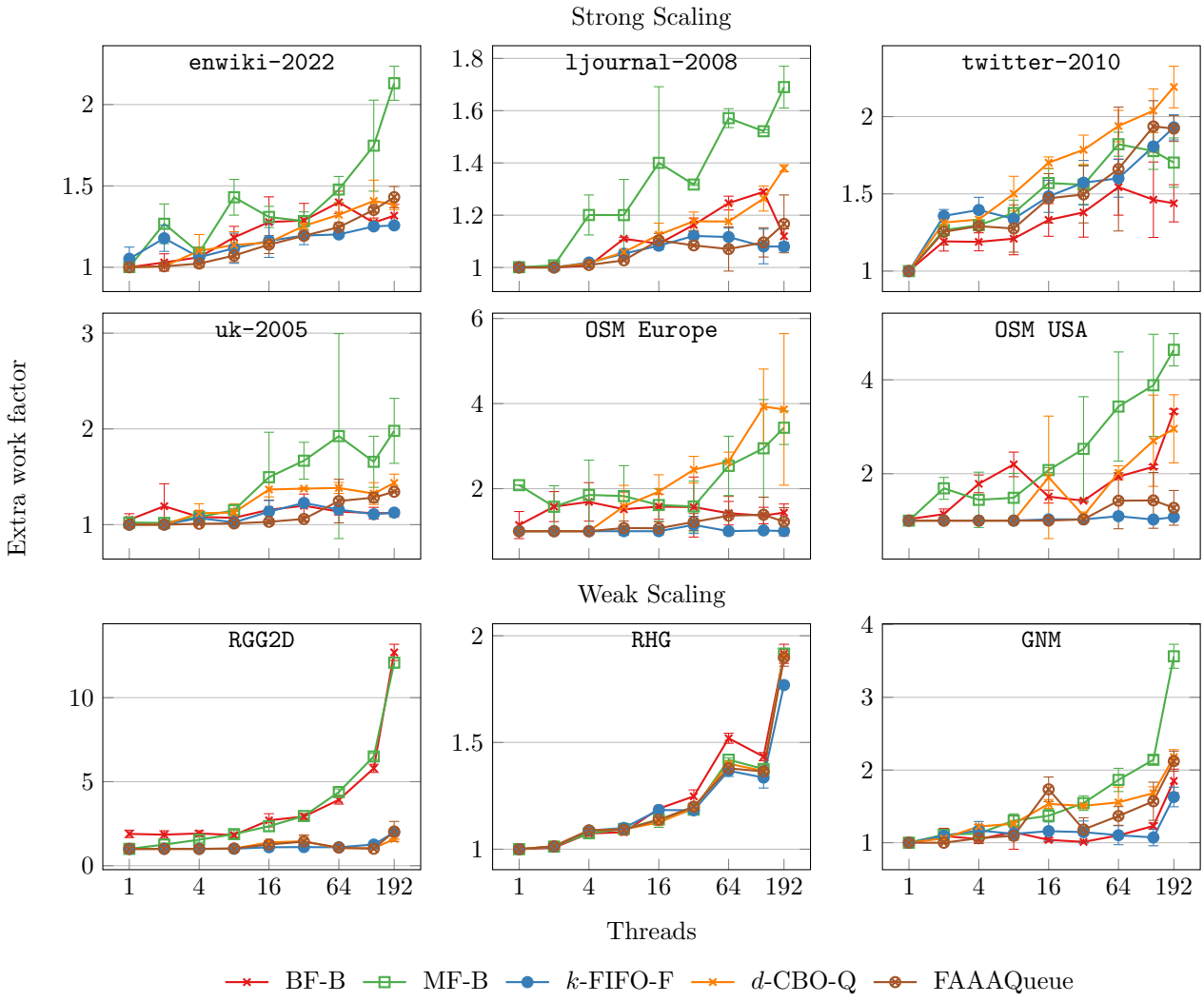


Figure 6.5: Comparing extra work of weak and strong scaling BFS benchmarks. Only the queue configurations from Figure 6.4 are shown.

**7 Conclusion and Future Work.** With the MultiFIFO and the BlockFIFO, we have introduced relaxed concurrent FIFOs that allow very high throughput and scale with increased relaxation. In particular the BlockFIFO still offers many opportunities for further improvement. Besides giving a "proper" theoretical analysis, we would like to (1) make them perform better when almost empty and (2) avoid the quadratic dependence of rank errors on the number of threads.

Problem (1) may be addressed by observing that, with very few elements, we can remove any element and we can insert a new element anywhere while respecting the rank error bounds. One possibility is to allow overlapping push and pop windows and allow concurrent insertions and deletions in the same block. However, this approach makes emptiness detection more difficult.

For Problem (2), we consider guiding threads to blocks more efficiently by augmenting the bitsets with a hierarchical structure supporting efficient, low-contention traversal. This might allow us to transition to blocks whose size need not scale with the number of threads.

We showed that relaxed FIFO queues offer a simple yet promising approach to parallel graph searches. We intend to investigate their application in further domains, and view a parallel version of the preflow-push algorithm for the maximum-flow problem [9] as a promising candidate. Analyzing the connection between the quality of a queue and the extra work incurred is another interesting direction for future research, which may help to select and tune the appropriate data structure for a given application.

Finally, we find it interesting to look at adaptations of the BlockFIFO for other architectures like GPUs or distributed memory.

**Acknowledgments.** This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882 500).


## References

[1] Y. Afek, G. Korland, and E. Yanovsky, *Quasi-Linearizability: Relaxed Consistency for Improved Concurrency*, in Principles of Distributed Systems, 2010, pp. 395–410, https://doi.org/10.1007/978-3-642-17653-1_29.

[2] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, *Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated*, ACM SIGPLAN Notices, 46 (2011), https://doi.org/10.1145/1925844.1926442.

[3] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, *Ubicrawler: A scalable fully distributed web crawler*, Software: Practice & Experience, 34 (2004), pp. 711–726.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Fourth Edition*, Apr. 2022.

[5] F. Ellen, D. Hendler, and N. Shavit, *On the inherent sequentiality of concurrent objects*, SIAM Journal on Computing, 41 (2012), pp. 519–536, https://doi.org/10.1137/08072646X, https://arxiv.org/abs/https://doi.org/10.1137/08072646X.

[6] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz, *Communication-free massively distributed graph generation*, in 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018, 2018.

[7] A. Gidenstam, H. Sundell, and P. Tsigas, *Cache-Aware Lock-Free Queues for Multiple Producers/ Consumers and Weak Memory Consistency*, in Principles of Distributed Systems, 2010, pp. 302–317, https://doi.org/10.1007/978-3-642-17653-1_23.

[8] A. V. Goldberg and C. Harrelson, *Computing the shortest path: A search meets graph theory.*, in SODA, vol. 5, 2005, pp. 156–165.

[9] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum-flow problem*, J. ACM, 35 (1988), pp. 921–940, https://doi.org/10.1145/48014.61051.

[10] A. Haas, T. Hütter, C. M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova, *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.

[11] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and

A. Sezgin, *Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation*, in Proceedings of the ACM International Conference on Computing Frontiers, CF '13, May 2013, pp. 1–9, https://doi.org/10.1145/2482767.2482789.

[12] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, *Quantitative relaxation of concurrent data structures*, SIGPLAN Not., 48 (2013), pp. 317–328, https://doi.org/10.1145/2480359.2429109.

[13] M. P. Herlihy and J. M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Transactions on Programming Languages and Systems, 12 (1990), pp. 463–492, https://doi.org/10.1145/78969.78972.

[14] M. Hoffman, O. Shalev, and N. Shavit, *The Baskets Queue*, in Principles of Distributed Systems, 2007, pp. 401–414, https://doi.org/10.1007/978-3-540-77096-1_29.

[15] C. M. Kirsch, M. Lippautz, and H. Payer, *Fast and Scalable, Lock-Free k-FIFO Queues*, in Parallel Computing Technologies, 2013, pp. 208–223, https://doi.org/10.1007/978-3-642-39958-9_18.

[16] H. Kwak, C. Lee, H. Park, and S. Moon, *What is twitter, a social network or a news media?*, in Proceedings of the 19th international conference on World wide web, 2010, pp. 591–600.

[17] M. M. Michael and M. L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, in Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '96, May 1996, pp. 267–275, https://doi.org/10.1145/248052.248106.

[18] A. Morrison and Y. Afek, *Fast concurrent queues for x86 processors*, in Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Feb. 2013, pp. 103–112, https://doi.org/10.1145/2442516.2442527.

[19] H. Rihani, P. Sanders, and R. Dementiev, *MultiQueues: Simple Relaxed Concurrent Priority Queues*, in Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, June 2015, pp. 80–82, https://doi.org/10.1145/2755573.2755616.

[20] R. Romanov and N. 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, PPoPP '23, Feb. 2023, pp. 14–26, https://doi.org/10.1145/3572848.3577485.

[21] A. Rukundo, A. Atalar, and P. Tsigas, *Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework*, in 33rd International Symposium on Distributed Computing ( DISC 2019), vol. 146 of Leibniz International Proceedings in Informatics (LIPIcs ), 2019, pp. 31:1–31:15, https://doi.org/10.4230/LIPIcs.DISC.2019.31.

[22] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, *A lock-free algorithm for concurrent bags*, in Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11, June 2011, pp. 335–344, https://doi.org/10.1145/1989493.1989550.

[23] P. Tsigas and Y. Zhang, *A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems*, in Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '01, July 2001, pp. 134–143, https://doi.org/10.1145/378580.378611.

[24] K. von Geijer, P. Tsigas, E. Johansson, and S. Hermansson, *Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue*, in Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP '25, Feb. 2025, pp. 382–395, https://doi.org/10.1145/3710848.3710892.

[25] S. Walzer and M. Williams, *A Simple yet Exact Analysis of the MultiQueue*, Feb. 2025, https://doi.org/10.48550/arXiv.2410.08714, https://arxiv.org/abs/2410.08714.

[26] M. Williams and P. Sanders, *Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues*, Apr. 2025, https://doi.org/10.48550/arXiv.2504.11652, https://arxiv.org/abs/2504.11652.

[27] M. Williams, P. Sanders, and R. Dementiev, *Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues*, in 29th European Symposium on Algorithms, ESA, Aug. 2021, https://doi.org/10.4230/LIPIcs.ESA.2021.81.

**A Lock-Free Ring Buffer Implementation.** Here, we describe the lock-free implementation of the BlockFIFO using a fixed-size ring buffer. The blocks are stored in a linear array $A$ of size $|A| = k \cdot w$ for some integer $k \geq 3$. Each block consists of a *header* and an array of $C$ *cells* that store the elements. The block header is represented as a single integer that encodes an epoch, a pop counter, a push counter, and a bit to indicate whether the block is claimed. The epoch is a monotonically increasing integer that is incremented whenever the block is closed. A *block index* is an integer that encodes both the position of a block in the array and its *epoch*. Specifically, block index $i$ references block $A[i \mod |A|]$ in epoch $\lfloor i/|A| \rfloor$. The block index $i = -1$ indicates an invalid block by convention. The push and pop windows are represented as block indices to the first block in the respective window. A block index is *valid* if it is not $-1$ and the referenced block has the same epoch as the block index. If the epoch of a block index is smaller than the epoch of the referenced block, the block is regarded as closed. Each thread maintains block indices to the blocks it last pushed into and popped from, respectively. Pseudocode for the `push` and `pop` operations is provided in Algorithm 2 and 3, respectively.

To claim a block, a thread changes the block's claim bit from `false` to `true`. Inserting an element into a claimed block involves three steps. First, the threads reads the block header and checks that the block index is valid and that the block is not full. Second, it attempts to place the element into the cell indicated by the push counter, checking that the cell was empty (i.e., contained $\perp$) prior to the operation. Third, the thread attempts to *commit* the element by incrementing the push counter in the header if the header remained unchanged during the operation. If committing fails, the operation is reverted by resetting the cell to $\perp$.

If any of the three steps fails, the thread attempts to claim another random unclaimed block in the push window. If the push window contains no unclaimed blocks, the thread advances the push window by $w$ blocks, or, if the queue is full, the insertion fails.

When deleting an element from a non-empty block, a thread first attempts to *reserve* the element. If it was the last element in the block, it does so by incrementing the epoch counter, thereby closing the block. Otherwise, it does so by incrementing the block's pop counter. After reserving an element, the thread atomically retrieves the element and replaces its cell with $\perp$.

Crucially, an inserted element cannot be deleted until it is committed: Deletions only consider cells with indices within the block's push and pop counters. Once the element is inserted into its cell, no other thread can advance the block's push counter beyond the cell's index, since subsequent insertions into that cell would fail. Moreover, if the push counter was already greater than the cell's index, then the pop counter must also have been greater than the cell's index; otherwise, the cell would not have been empty when the element was inserted. If a thread suspends after writing an element to a cell but before committing it, only the cells up to this cell can be used in later epochs. When the thread resumes, it will either commit the element or revert the operation, making all cells in the block available again. As soon as a deleting thread reserves an element in a block, no other thread can reserve the same element. If a deleting thread suspends right after reserving an element but before deleting it, the remaining elements in the block can still be deleted and the block can still be closed. Again, only the cells up to the reserved element can be used in later epochs. When the thread resumes, it will delete the element, making all cells in the block available again.

**A.1 Lock-Freedom.** We sketch the main argument for lock-freedom, showing that within a bounded number of steps, at least one thread is guaranteed to complete its operation.

An inserting thread will, within a bounded number of steps, either complete its operation, attempt to insert an element into a claimed block, attempt to claim a new block, or attempt to advance the push window. When inserting an element succeeds, the operation completes in a bounded number of steps. If inserting an element into a claimed block fails, it attempts to claim a new block. Since each block can be claimed at most once per epoch, one inserting thread will, within a bounded number of steps, either complete the insertion (potentially failing) or successfully advance the push window. Once the push window is advanced, all blocks in the new push window are unclaimed. The pop window cannot advance to these blocks before at least one insertion into a block in the new push window is completed. Since insertions never update the header or place elements in the cells of a block claimed by other threads, at least one thread must be able to complete its operation within a bounded number of steps.

A deleting thread will, within a bounded number of steps, either complete its operation, attempt to reserve an element to delete, attempt to close a block, or attempt to advance the pop window. When reserving an element succeeds, the operation completes in a bounded number of steps. If reserving an element or closing a block fails, it is retried until it succeeds or the block is closed. However, the block header can be changed at most $C + 1$ times by inserting threads before a deletion

**Algorithm 2:** Pseudocode for the `push` operation.

```
Result: true if the insertion succeeds; false if the queue is full
Data: pushBlock, the block index of the last block the thread inserted into
Input: Element to insert e

Function push(e)
    p ← load(pushWindow)
    if pushBlock ≥ p then
        h ← load(A[getPos(pushBlock)].header)
        if h.epoch = getEpoch(pushBlock) ∧ insertInBlock(h, pushBlock, e) then
            if h.pushes = C − 1 then pushBlock() ← −1
            return true
    repeat
        r ← random(0, w − 1)
        for j ← 0 to w − 1 do                                  ▷ search random unclaimed block
            i ← p + (r + j  mod w)
            h ← load(A[getPos(i)].header)
            h* ← toHeader(getEpoch(i), 0, 0, false)
            h′ ← toHeader(getEpoch(i), 0, 0, true)
            if h = h* ∧ CAS(A[getPos(i)].header, h, h′) then                    ▷ block claimed
                if insertInBlock(h, i, e) then pushBlock ← i; return true
        if p + w − load(popWindow) = |A| then                                  ▷ queue is full
            pushBlock ← −1; return false
        p ← CAS(pushWindow, p, p + w)                                ▷ advance push window

Function insertInBlock(h, i, e)
    if CAS(A[getPos(i)].cells[h.pushes], ⊥, e) then
        h′ ← toHeader(h.epoch, 0, h.pushes + 1, true)
        if CAS(A[getPos(i)].header, h, h′) then return true
        A[getPos(i)].cells[h.pushes] ← ⊥
    return false
```

**Algorithm 3:** Pseudocode for the pop operation.

```
Result: The deleted element e, or ⊥ if the queue is empty.
Data: popBlock, the block index of the last block the thread deleted from

Function pop()
    if popBlock ≠ −1 then
        h ← load(A[getPos(popBlock)].header)
        if h.epoch = getEpoch(popBlock) ∧ reserveElement(h, popBlock) then
            return swap(A[getPos(popBlock)].cells[h.pops], ⊥)
    repeat
        p ← load(popWindow); q ← load(pushWindow)
        if p + w < q ∧ load(A[getPos(p)].header).epoch ≠ getEpoch(p) then
            CAS(popWindow, p, p + 1); continue                    ▷ first block is closed
        r ← random(0, w − 1)
        for j ← 0 to w − 1 do
            i ← p + (r + j  mod w)
            repeat
                h ← load(A[getPos(i)].header)
                if h.epoch ≠ getEpoch(i) then break
                if reserveElement(h, popBlock) ∧ h.pushes > 0 then
                    popBlock ← i; return swap(A[getPos(i)].cells[h.pops], ⊥)
        if p + w = q then                          ▷ pop window is directly behind push window
            pushWindowEmpty ← true
            for i ← q to q + w − 1 do                          ▷ scan push window for elements
                h ← load(A[getPos(i)].header)
                if h.pushes > 0 then pushWindowEmpty ← false; break
            if pushWindowEmpty ∧ q = load(pushWindow) then return ⊥
            CAS(pushWindow, q, q + w)                                  ▷ advance push window
            CAS(popWindow, p, p + w)                                    ▷ advance pop window

Function reserveElement(h, i)
    if h.pops + 1 < h.pushes then                    ▷ block has more than one elements
        h′ ← toHeader(h.epoch, h.pops + 1, h.pushes, true)
    else                                                               ▷ block can be closed
        h′ ← toHeader(h.epoch + 1, 0, 0, false)
    return CAS(A[getPos(i)].header, h, h′)
```

successfully reserves an element or closes the block. Since each block can be closed at most once per epoch, one deleting thread will, within a bounded number of steps, either complete its operation (potentially failing) or successfully advance the pop window. Thus, if at least one element is in the queue, after a bounded number pop window advances, at least one block in the pop window will contain an element. After a bounded number of steps, at least one deleting thread will be able to reserve an element in the pop window and complete its operation.

**B Bitset Details.** The bitset is partitioned into small atomic units. In order to prevent false sharing, atomic units are aligned to cache lines. It is desirable to establish the bitset as an arbiter of truth to actually allow for avoiding potentially numerous block header reads. This necessitates that the bitset must only exhibit one-sided errors, where a bit may be set even if the corresponding block does not contain any elements. Some kind of one-sided error is unavoidable, as block push/pop operations must be ordered in some way with the corresponding bitset modifications. By choosing the side of the errors like we have, it enables deleters to concern themselves only with the set bits, which they eventually reset over a bounded number of delete operations. Blocks associated with unset bits are of no concern to them. Additionally, it is necessary to bundle an epoch with the bitset, which can be stored and modified alongside each atomic unit. Without epochs, deleters may create a two-sided error by resetting the bit of a block that contained no elements when they have started the operation, but has since been filled in the subsequent epoch.

The search operation on each atomic unit can be efficiently implemented using a fixed amount of instructions independently of the size of the atomic unit $u$. This implementation relies on the bitwise rotation and count leading zeroes instructions commonly available on CISC architectures.[10] The search operation over the entire bitset simply encompasses a linear iteration over all atomic units, executing the atomic unit search operation on each until a desired bit is found.

This means that $u$ is a tuning parameter, offering a trade-off between reducing the contention on the individual atomic units with a small $u$ and allowing more blocks to be checked in the same amount of operations with large $u$. In practice, choosing $u$ to be minimal has shown to provide the best performance until very high degrees of relaxation, where contention becomes so low that the reduced operation count proves more beneficial.

[10] For example `ROR` and `LZCNT` on `x86`.

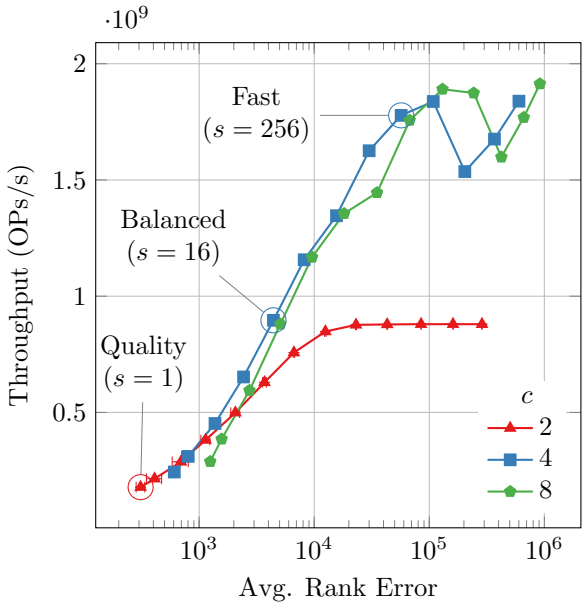


Figure C.1: MultiFIFO parameter tuning of the subqueues per thread $c$ and stickiness period $s$ using the push-pop benchmark. Stickiness ranges from 1 to 4096. Annotated data points are configurations used in further experiments.

**C Parameter Tuning.** We investigate the trade-off between quality and performance offered by different queues and their configurations and select specific configurations for further experiments. For this purpose, we use the push-pop benchmark with the maximum of 192 threads on machine `AMD` to measure both throughput and quality.

*MultiFIFO.* In Figure C.1, we compare different configurations of the MultiFIFO. We vary the subqueues per thread $c$ between 2, 4 and 8 and the stickiness period between 1 (which is equivalent to no sticking behavior) and 4096, only considering values that are a power of two.

The MultiFIFO plateaus relatively quickly with increasing stickiness, offering no better performance at continuously decreasing quality. Doubling the queues per thread from 2 to 4 doubles the achievable performance, however, further doubling the queues per thread from 4 to 8 has no beneficial effect on either performance or quality.

*BlockFIFO.* In Figure C.2 we compare different configurations of the BlockFIFO. We tune the parameters $B$ and $C$. Only powers of two are tested, with $B$ ranging from 1 to 16 and $C$ ranging from 7 to 2047.

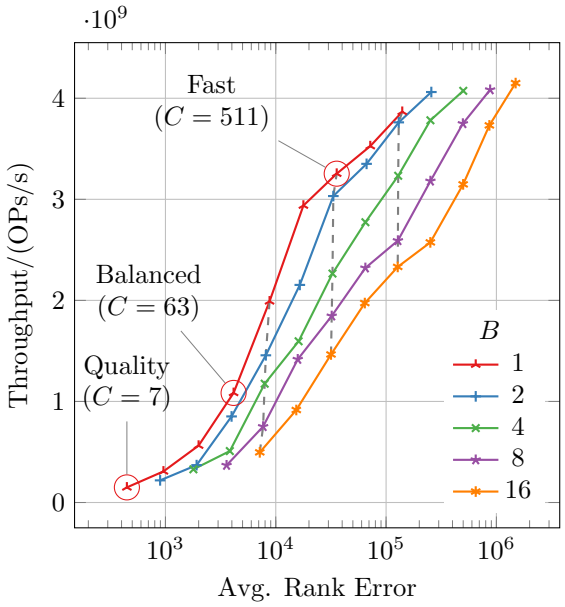


Figure C.2: BlockFIFO parameter tuning of the blocks per window per thread $B$ and cells per block $C$ using the push-pop benchmark. Annotated data points are configurations used in further experiments. The gray dashed lines indicate configurations with the same total number of cells in the windows (i.e., the product of $B$ and $C$ is the same).

Notably, $C$ is always $2^x - 1$ in order to accommodate for the 64-bit block header.

The BlockFIFO exhibits a peculiar vertical clustering of points around one level of quality as illustrated by the dashed lines. This behavior can be sufficiently explained by quality correlating to the amount of *cells* per window, so the product of window size $w$ and block size $C$.[11] For a fixed amount of cells per *window* a low $B$ and high $C$ appears optimal. Such a configuration minimizes the amount of potentially high-contention block claim operations necessary to fill/empty an entire window, while maximizing the time that can be spent in cache-friendly, typically low-contention inner-block operations. Additionally, a low $B$ minimizes the maximum possible time spent looking for a valid block within a window by minimizing the amount of blocks there are.

**D Multi-Start BFS.** On graph instances where scalability is limited, more significant speedups can be achieved by executing multiple searches concurrently within a single queue. We start searches from $s$ nodes, storing $s$ distinct distance arrays and including the associated source node index in the elements pushed to the queues. This is reminiscent to the pre-computation required for an $A^*$ with landmarks (ALT) search [8] without weights.

As shown in Figure D.1, the scalability is vastly superior to a single search being executed. Especially the MultiFIFO and BlockFIFO benefit from the additional work, achieving a speedup of two orders of magnitude with the highest number of threads. While the $d$-CBO outperformed the BlockFIFO and MultiFIFO on the `RGG2D` graph on the standard BFS benchmark, this is not the case for the multi-start BFS.

**E Complete Benchmarks.**

*Quality.* Figure E.1 shows the rank error behavior of the relaxed competitors on the push-pop benchmark for different thread counts. All competitors demonstrate the expected linear scaling of rank errors for all competitors.

*Producer-Consumer on other Machines.* The behavior is largely similar on the other machines, as seen in Figure E.2. The BlockFIFO performance degrades on the Intel machine, with a stark drop in performance for producer-heavy workloads for the fast BlockFIFO that is also similarly displayed by the fast MultiFIFO.

*BFS with all competitors.* Figure E.3 and Figure E.4 show the BFS benchmarks with all competitors and configurations.

[11] There are always $w$ less cells per window due to the block headers, but that is irrelevant in practice.

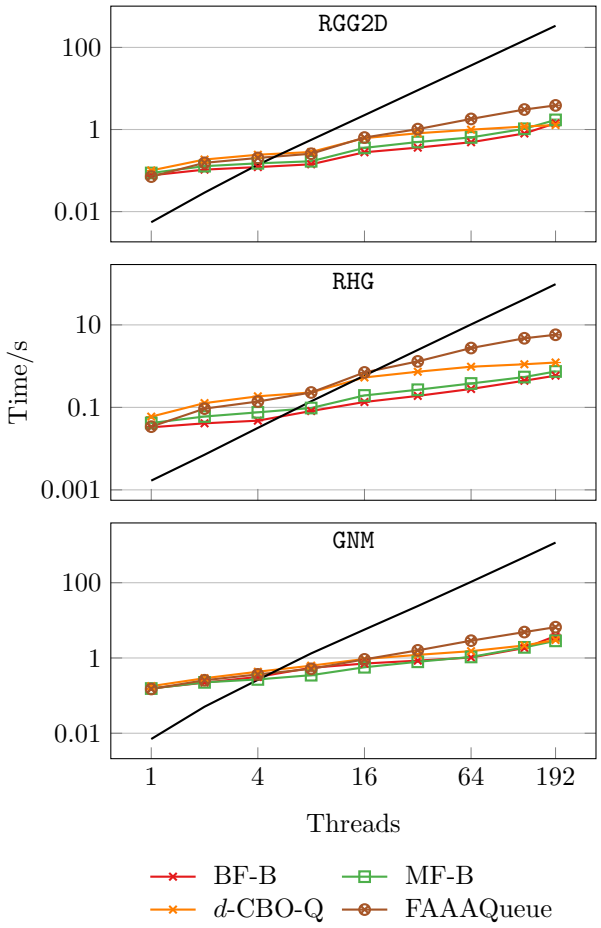


Figure D.1: Weak scaling BFS benchmarks with $s = 8$ concurrent searches per instance. Graph instances are the same as used in Figure 6.4. The $k$-FIFO is unable to complete this benchmark due to its inability to store a full 64 bits of data in its elements which is required to account for the source node indices.

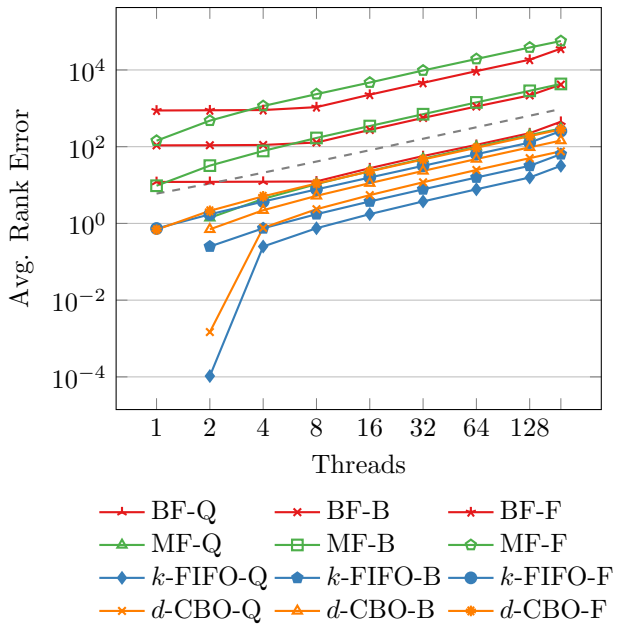


Figure E.1: The rank errors during the push-pop benchmark at different thread counts. Some configurations exhibit no rank errors with a single thread, these data points are omitted. For reference, we include the linear function $5p + 1$ as a dashed line.

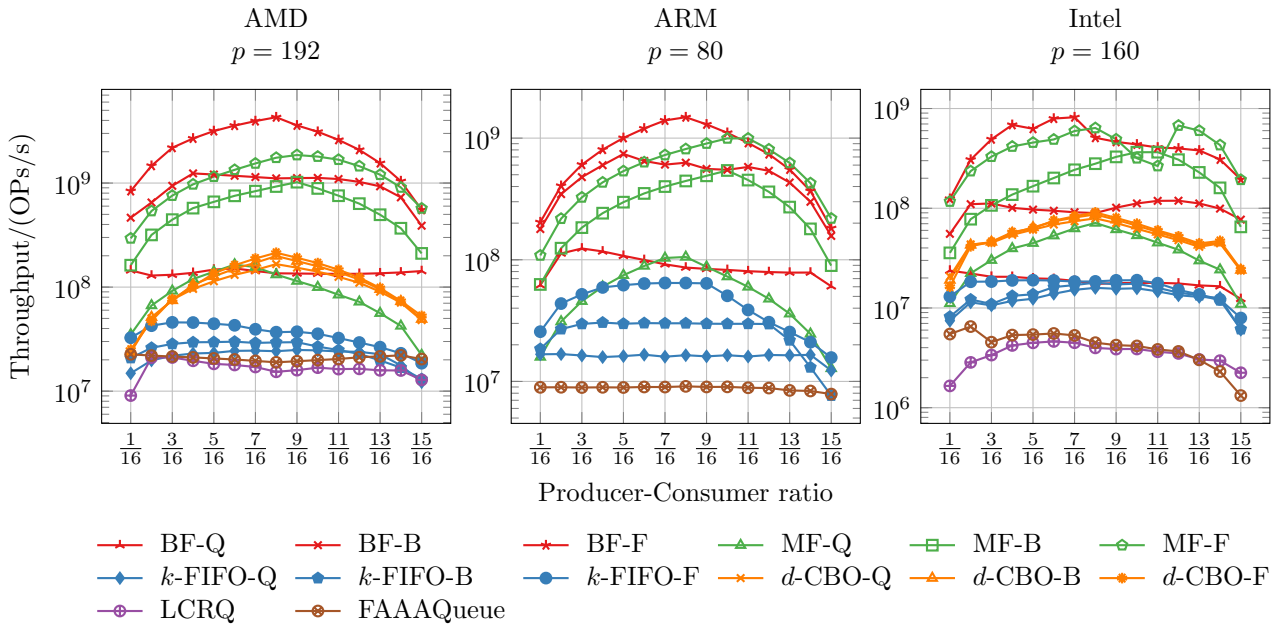


Figure E.2: Measuring throughput with different producer-consumer ratios on other machines.

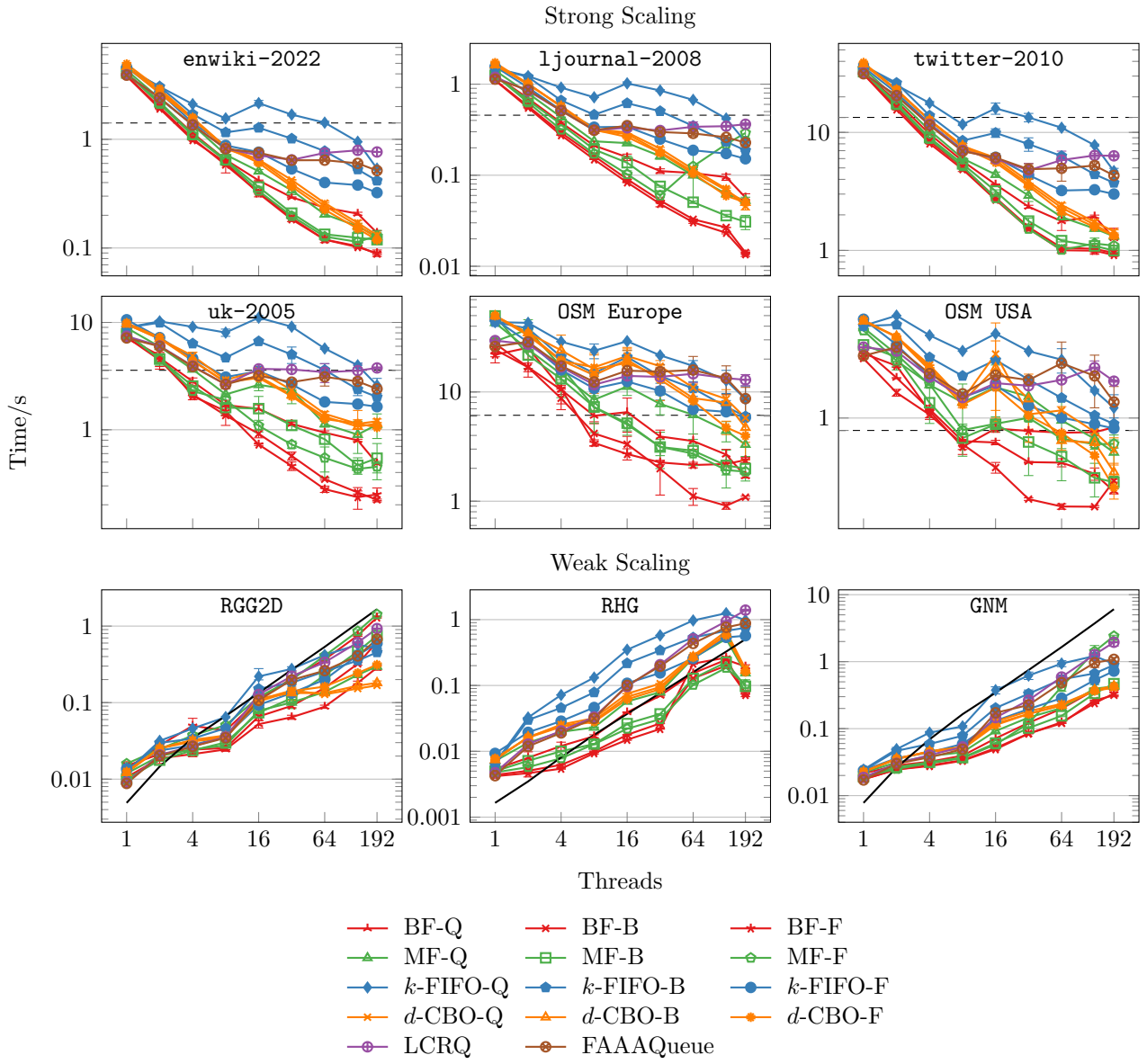


Figure E.3: Equivalent to Figure 6.4 with all queues and configurations shown.

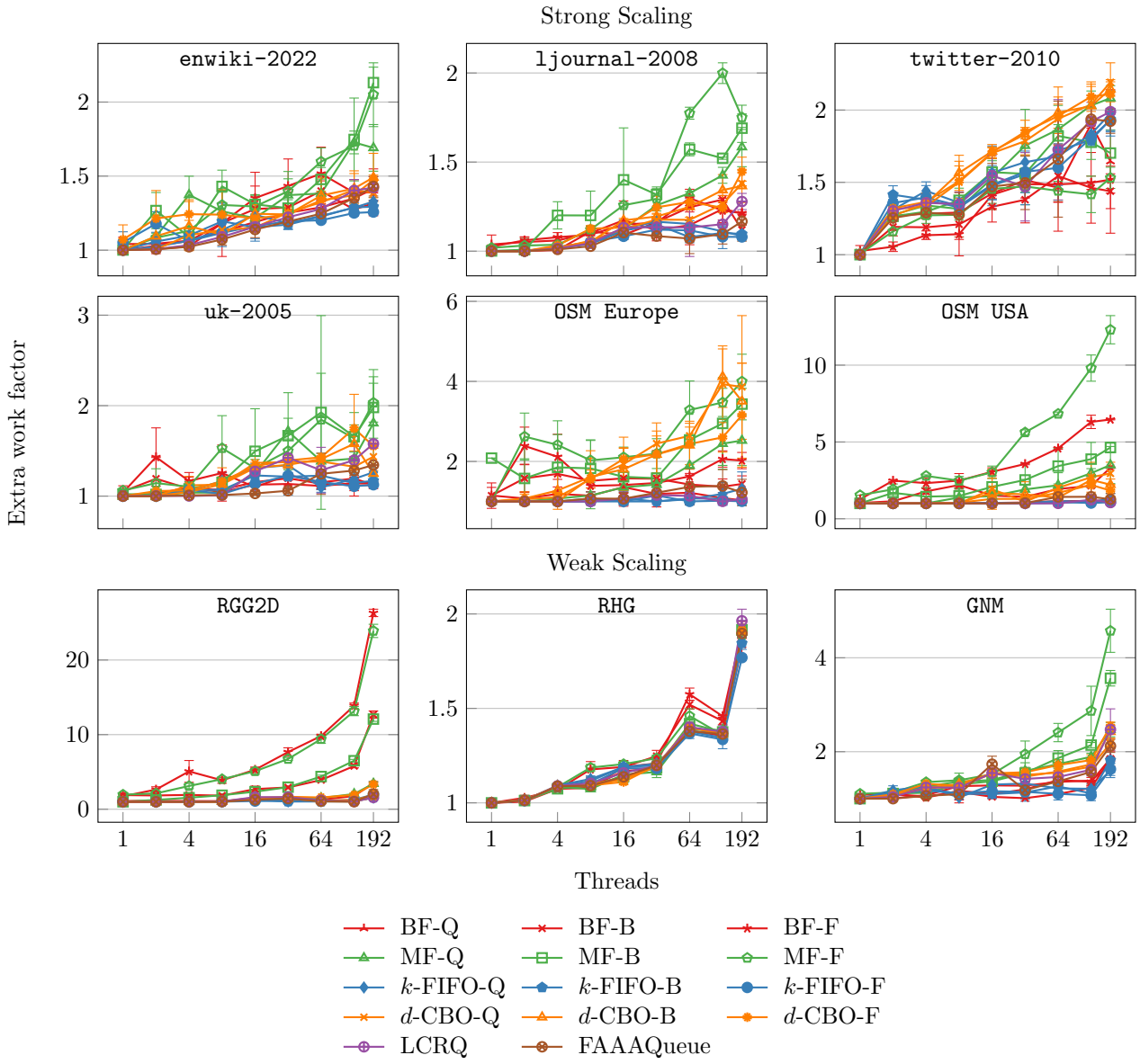


Figure E.4: Equivalent to Figure 6.5 with all queues and configurations shown.