BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling

Jiawei Wang[1,2,3], Diogo Behrens[1,2], Ming Fu[1,2,*], Lilith Oberhauser[1,2], Jonas Oberhauser[1,2], Jitang Lei[1,2], Geng Chen[2], Hermann Härtig[3], and Haibo Chen[2,4]

[1]*Huawei Dresden Research Center* [2]*Huawei OS Kernel Lab*
[3]*Technische Universität Dresden* [4]*Shanghai Jiao Tong University*

Abstract

Concurrent bounded queues have been widely used for exchanging data and profiling in operating systems, databases, and multithreaded applications. The performance of state-of-the-art queues is limited by the interference between multiple enqueues (enq-enq), multiple dequeues (deq-deq), or enqueues and dequeues (enq-deq), negatively affecting their latency and scalability. Although some existing designs employ optimizations to reduce deq-deq and enq-enq interference, they often neglect the enq-deq case. In fact, such partial optimizations may inadvertently increase interference elsewhere and result in performance degradation.

We present Block-based Bounded Queue (BBQ), a novel ringbuffer design that splits the entire buffer into multiple blocks. This eliminates enq-deq interference on concurrency control variables when producers and consumers operate on different blocks. Furthermore, the block-based design is amenable to existing optimizations, *e.g.*, using the more scalable *fetch-and-add* instruction. Our evaluation shows that BBQ outperforms several industrial ringbuffers. For example, in single-producer/single-consumer micro-benchmarks, BBQ yields 11.3x to 42.4x higher throughput than the ringbuffers from Linux kernel, DPDK, Boost, and Folly libraries. In real-world scenarios, BBQ achieves up to 1.5x, 50.5x, and 11.1x performance improvements in benchmarks of DPDK, Linux io_uring, and Disruptor, respectively. We verified and optimized BBQ on weak memory models with a model-checking-based framework.

1 Introduction

Concurrent bounded queues are pervasive in operating systems, databases, and multithreaded applications. They transport data, distribute work, and are used to profile and decouple components. Their performance is crucial for achieving highly scalable and low-latency operation of numerous systems.

The main factor determining performance of a queue is the interference between concurrent operations, *i.e.*, between enqueues, between dequeues, or between enqueues and dequeues. We refer to these as *enq-enq*, *deq-deq*, and *enq-deq* interference, respectively. Interferences manifest in the form of 1) cache-line bouncing when control variables are frequently updated by one thread and read by another, *e.g.*, to check if the queue has data, and 2) serialization of contended updates to control variables, *e.g.*, when multiple threads try to create or read the same entry. Existing queue designs often employ optimizations to reduce enq-enq and deq-deq interference, *e.g.*, updating control variables with "always-successful" atomic instructions such as *fetch-and-add* (`FAA`) [14, 40, 41, 47] because, in principle, they can be serialized in hardware and thus perform better under high contention than software solutions with *compare-and-swap* (`CAS`) [38, 43]. However, existing designs tend to neglect the enq-deq interference even though it substantially impacts performance, in particular in the common single producer or single consumer scenarios, *e.g.*, ringbuffers for asynchronous I/O in Linux io_uring [13].

In fact, some queue optimizations from the literature [38, 40] inadvertently increase the enq-deq interference or introduce undesirable side-effects that degrade performance in uncontended cases. For example, dequeue operations using `FAA` must either block in a pessimistic way [14], or risk overtaking slow concurrent enqueue operations; to avoid data corruption, such enqueue operations must be invalidated and repeated later [40]. Besides harming performance, such strategies cannot be applied for online profiling where enqueues writing a log should not be delayed by dequeues that read the log. Similarly, some techniques that avoid concurrent enqueue operations from waiting for each other also require dequeue operations to invalidate parts of the queue [38]. Strategies to improve performance of these techniques by reducing the number of invalidations, *e.g.*, busy-looping before invalidating [38, 40], drastically increase the latency of dequeue calls on empty queues, making them unsuitable for certain workloads, *e.g.*, multiplexing across multiple message queues.

We present Block-based Bounded Queue (BBQ), a novel ringbuffer design that dramatically reduces the enq-deq interference by splitting the entire buffer into multiple blocks and splitting the control variables into the block-level and queue-level variables. In the common case, enqueue and de-

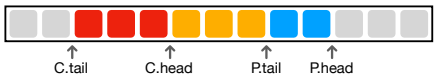
*Ming Fu (ming.fu@huawei.com) is the corresponding author.

queue only access block-level control variables of their current blocks. When enqueue and dequeue work on different blocks, the disjoint control variables avoid any interference between these operations. That is particularly important in reducing the cache-line bouncing of head and tail pointers when determining whether the queue is full or empty. Furthermore, we use hardware-serialized FAA operations to update block-level control variables for allocating entries inside blocks, while queue-level control variables on the other hand are updated with slower, software-serialized CAS operations; since this is only necessary in the rare event that operations need to move to the next block, the performance impact of these operations is negligible. Our block-based approach allows us to perform these optimizations without incurring undesirable side-effects. Finally, to ensure that BBQ correctly works on weak memory models (WMMs) — including those from Arm [2] and RISC-V [28] architectures — we have verified and optimized the barriers and fences of BBQ with the VSYNC framework [42].

In contrast to previous work, our block-based approach is applicable to a large spectrum of scenarios. BBQ supports single or multiple producers/consumers, fixed- or variable-sized entries, and retry-new and drop-old modes. Retry-new is the typical producer-consumer mode for message passing and work distribution scenarios; drop-old is a lossy/overwrite mode for profiling/tracing [5, 24] and debugging [44] scenarios, in which producers may overwrite unconsumed data if the buffer is full.

In our experimental evaluation, BBQ outperforms several industrial queues and ringbuffers. In single-producer/single-consumer micro-benchmarks, BBQ yields 11.3x to 42.4x higher throughput than Linux circular buffer [22], DPDK ring buffer [9], Boost lock-free queue [4], and Meta's Folly queue [14]. In real-world scenarios, BBQ achieves up to 1.5x, 50.5x, and 11.1x performance improvements in benchmarks of DPDK, Linux io_uring [13], and LMAX Disruptor [23], respectively. In our profiling benchmarks, BBQ enabled with the lossy/overwrite mode achieves up to 4.7x higher throughput than Google's Guava EvictingQueue [15] and Apache Commons CircularFifoQueue [1], and can sustain up to 143.2x lower enqueue latency than the other two queues.

The remainder of this work is organized as follows. In Sec. 2, we gradually introduce the challenges of reducing the interference between enqueue and dequeue operations, discussing how existing queues tackle these challenges, and the limitations of their solutions. In Sec. 3, we present our block-based approach and the high-level design of BBQ. In Sec. 4, we describe BBQ implementation including the support for retry-new and drop-old modes and variable-sized entries. In Sec. 5, we report our results in verifying BBQ on WMMs and relaxing its memory barriers. In Sec. 6, we experimentally compare the performance of BBQ and several industry-grade concurrent queues. In Sec. 7, we conclude our work.




```
1  enqueue(data){
2  again:
3    ph = LOAD(P.head);
4    pn = ph + 1;
5    if (pn > LOAD(C.tail) + SZ)
6      return FULL;
7    if (!CAS(P.head, ph, pn))
8      goto again;
9    entry[pn % SZ] = data;
10   while(LOAD(P.tail) != ph);
11   STORE(P.tail, pn);
12   return OK;
13 }
```

```
14 dequeue(){
15 again:
16   ch = LOAD(C.head);
17   cn = ch + 1;
18   if (cn > LOAD(P.tail))
19     return EMPTY;
20   if (!CAS(C.head, ch, cn))
21     goto again;
22   data = entry[cn % SZ];
23   while(LOAD(C.tail) != ch);
24   STORE(C.tail, cn);
25   return data;
26 }
```

Figure 1: A simple MPMC bounded queue. CAS, LOAD, and STORE are atomic operations with sequentially consistent semantics on WMMs. C.head and C.tail refer to consumers; P.head and P.tail refer to producers.

2 Background and Related Work

We now introduce scalability challenges of bounded queue designs and discuss related work and their limitations. Figure 1 illustrates the discussion in this section depicting a simple lockless bounded queue with multiple-producer multiple-consumer (MPMC) support, which is the algorithm behind the widely-used DPDK ringbuffer [9].

Producers first check whether the queue is full (Line 5) and then try to *allocate* the next entry via CAS (Line 7). Upon success, the producer copies the data into the entry and *commits* it (Line 11). Similarly, consumers first try to *reserve* an entry (Line 20). Upon success, the consumer copies the data back and confirms that the data has been *consumed* (Line 24).

(P1) Consumer contention on C.head. A straightforward form of deq-deq interference is caused by multiple consumers concurrently calling dequeue and contending on updates to C.head. Several works (on bounded and unbounded queues) tackle this contention using FAA to update the head [38,40,43] because FAA is more scalable than CAS on common architectures. However, since FAA cannot conditionally update the memory location, it may break, for example, the invariant of C.head never exceeding P.tail. To address that, Meta's Folly queue [14] implements the partial, not total dequeue method, which spins until dequeue succeeds [33]. With such interface, dequeue calls return only if there is an entry to consume, otherwise blocking the thread indefinitely.

Another solution is to "fix the state" when C.head exceeds P.head by invaliding entries between them, as done by LCRQ [38]. Unfortunately, that causes consumers to hamper the progress of producers. SCQ [40] solves the producer starvation by limiting the number of consecutive invalidations with a threshold, as we describe below. Nevertheless, the remaining invalidations still degrade the enqueue performance, and the SCQ implementation [39] employs a trick

to reduce the probability of invalidating entries: Consumers check several times in a loop whether the entry has been committed before actually invalidating it. Unfortunately, this *delayed invalidation* trick increases the latency of dequeue when the queue is empty by several orders of magnitude — we experimentally demonstrate this empty-deq in Sec. 6.2.3.

(P2) Producer contention on P.head. A straightforward form of enq-enq interference is caused by multiple producers concurrently calling enqueue and contending on updates to P.head. Here, Folly queue again resorts to FAA and turns enqueue into a partial method, which waits until free entries are available, potentially blocking the thread forever.

Nikolaev proposes a novel idea to implement a total queue while using FAA [40]: SCQD combines two SCQ index queues (fq and aq) with a data array. Upon enqueue on SCQD, the thread gets an index from fq, copies the data in the corresponding entry of the data array, and then puts the index into aq. Dequeueing works the other way around. The index queues are total on dequeue but partial on enqueue, *i.e.*, dequeue returns EMPTY if the queue is empty, whereas enqueue loops until it succeeds. Nevertheless, the combined SCQD is still total since index queues are never full, *i.e.*, the number of indexes is fixed and matches the maximum size of the index queues. Besides the constant overhead introduced by index queues, the high latency caused by the empty-deq issue in each SCQ index queue translates into high latency in SCQD for both empty-deq and full-enq cases (see Sec. 6.2).

(P3) Delayed P.tail and C.tail updates. Another typical enq-enq or deq-deq interference arises from the *in-order* policy to commit (resp. consume) entries — the default policy in DPDK ringbuffer. The head/tail mechanism, which resembles a ticket lock, brings issues analogous to Lock-Holder- and Lock-Waiter-Preemption [45]. For example, the preemption of a thread that is about to update P.tail (resp. C.tail) causes other enqueue calls (resp. dequeue calls) to uselessly spin (Lines 10 and 23) for arbitrary time periods.

Several queues implement, instead, *out-of-order* policies, allowing producers (resp. consumers) to commit (resp. consume) entries independently. In LCRQ, once consumers increment C.head such that it reaches P.tail, they start invalidating entries until C.head reaches P.head. That prevents producers from committing entries at indexes preceding C.tail, ensuring linearizability [34]. This approach starves producers and even livelocks the queue. For example, a consumer in an ongoing dequeue invalidates an entry when C.head $=$ P.tail $<$ P.head; the producer in the ongoing enqueue increments P.head to retry; the consumer realizes C.head still did not reach P.head and retries consuming, potentially invalidating the new entry if not yet committed; and so on.

SCQ uses a threshold T to restrict the number of consecutive entries invalidated, and, thus, avoid livelocks. When a consumer invalidates an entry, it atomically decrements T. A successful enqueue resets T to its initial constant $3n-1$, where n is twice the queue capacity. This constant is carefully derived to guarantee linearizability is never violated [40]. However, it introduces additional contention among producers and consumers updating the threshold variable, which has to be again mitigated by delayed invalidation.

DPDK [9] ringbuffer implements a more practical out-of-order policy called RTS mode [27], which trades linearizability to avoid invalidations. Consumers never move C.tail forward if C.tail would reach P.tail, returning EMPTY despite of any committed entry between P.tail and P.head; thus, violating linearizability. Producers employ the reciprocal strategy.

To enable out-of-order commits, RTS records whether all entries between P.tail and P.head are committed. The *prohibited window* between P.tail and P.head has dynamic length because P.tail is moved forward only once the last producer writing between P.tail and P.head commits. If producers would keep allocating entries, they would keep incrementing P.head and extending the prohibited window up to the total capacity of the queue. To prevent consumers from starving, RTS sets up a threshold to limit the maximum distance between P.tail and P.head. If that distance is reached, enqueue blocks until all producers between P.tail and P.head have committed. RTS enables out-of-order consumes with the recipocral approach.

(P4) Causes of enq-deq interference. There are two sources of interference between enqueues and dequeues: algorithmic and cache-related. While focusing on enq-enq or deq-deq cases, previous techniques introduce algorithmic interferences between enqueue and dequeue, *e.g.*, requiring producers and consumers to retry operations, increasing latency, potentially causing thread starvation, or even livelock.

Let us again consider the simple algorithm of Fig. 1. Even though cache misses caused by writing or reading the data cannot be eliminated, cache misses on the control variables are relevant. Every time a producer calls enqueue, it allocates an entry and increments P.tail. Every time a consumer calls dequeue, it potentially suffers a cache miss by reading P.tail. If the producer is far ahead the consumer, the cache misses at P.tail seem unjustifiable. Similarly, the producer suffers cache misses on C.tail even when there is plenty space in the queue between producers and consumers. In contrast to enq-enq and deq-deq, enq-deq interference is relevant even to the single-consumer/single-producer scenario, an important scenario for the industry. In Sec. 6, we experimentally show a correlation between a strong decrease of L1 cache misses and the performance improvements of BBQ (Sec. 3 and 4).

3 Design of BBQ

3.1 The Block-based Approach

BBQ splits the ringbuffer into blocks, as shown in Fig. 2. Each block contains one or more entries, usually multiple, depending on the configuration. The queue control variables are also split into queue-level and block-level variables. Control variables C.head and P.head now point to blocks instead

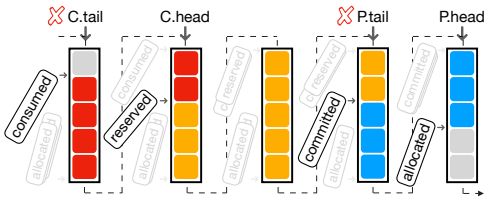


Figure 2: Block-based bounded queue (BBQ).

of entries; P.tail and C.tail are unnecessary for the algorithm. The block-level control variables include four cursors called `allocated`, `committed`, `reserved`, and `consumed`, which track the corresponding actions within each block.

The block-based approach greatly reduces the enq-deq interference. After a block is fully committed, its producer cursors (`allocated`/`committed`) remain unmodified until the block is fully consumed, causing no additional cache misses for consumers. Moreover, producers can always determine whether a block is fully allocated without accessing consumer cursors (`reserved`/`consumed`).

Multiple producers in the same block still contend on `allocated` and `committed`. Fortunately, the block-based approach enables the enqueue operation to use `FAA`, avoids costly invalidations, and still allows for a total method. Producers start using a block only once it has been fully consumed. Therefore, inside the block, `FAA` never allocates an entry that is not consumed yet, allowing enqueue to be total. `FAA` may make `allocated` out of the bound of the block, but state fixing is not required. Since each block has its own control variables, an out-of-bound cursor in one block does not affect the following block.

Although our dequeue operation employs `CAS` to avoid consumers from invalidating entries currently used by producers, BBQ still achieves similar or better performance than other designs with `FAA`-based dequeues. To further improve performance for machines with Armv8.1 [2] processors supporting Large System Extensions [3] (LSE), BBQ uses the atomic maximum instruction `MAX` instead of `CAS` in dequeue.

Finally, the block-based approach enables a practical out-of-order policy similar to RTS mode of DPDK. In this case, instead of updating control variables with double-width `CAS`, BBQ employs more scalable `FAA` and `MAX` instructions.

3.2 BBQ from a Bird's-eye View

In this section, we describe the high-level algorithm of BBQ, as shown in Fig. 3.

Producers. To enqueue data, a producer first retrieves the current value of P.head and the corresponding block identifier (Line 4). Next, it tries allocating an entry in the block (Line 5). If successful, the producer writes the data into entry `ety` and commits it (Line 7). The allocation fails if the block has already been fully allocated (Line 9). In this case, the

```
1  status := OK(T) | FULL | EMPTY | BUSY
2  status BBQ<T>::enqueue(T data){
3  loop:
4    (ph, blk) = get_phead_and_block();
5    switch (allocate_entry(blk)){
6      case allocated(ety):
7        commit_entry(ety, data);
8        return OK();
9      case BLOCK_DONE:
10       switch (advance_phead(ph)){
11         case NO_ENTRY: return FULL;
12         case NOT_AVAILABLE: return BUSY;
13         case SUCCESS: goto loop;
14       }
15   }
16 }
17 status BBQ<T>::dequeue(){
18 loop:
19   (ch, blk) = get_chead_and_block();
20   switch (reserve_entry(blk)){
21     case reserved(ety):
22       data = consume_entry(ety);
23       if (data != NULL) return OK(data);
24       else goto loop;
25     case NO_ENTRY: return EMPTY;
26     case NOT_AVAILABLE: return BUSY;
27     case BLOCK_DONE(vsn):
28       if (advance_chead(ch, vsn)) goto loop;
29       else return EMPTY;
30   }
31 }
```

Figure 3: High-level design of BBQ.

producer tries to advance P.head to the next block (Line 10). If successful, the producer jumps back to the `loop` label and retries the allocation (Line 13). In retry-new mode, advancing P.head fails if the next block is not yet fully consumed, *i.e.*, the whole queue is considered full. BBQ distinguishes the failure reason: `BUSY` when some dequeue operation is ongoing and `FULL` otherwise. Returning `BUSY` allows for custom backoff implementations at the caller side, *e.g.*, parking threads after a number of retries. In drop-old mode, advancing P.head does not fail except for a seldom case discussed in Sec. 4.3, for which `BUSY` is returned.

Consumers. The dequeue operation is somewhat analogous to enqueue. The consumer starts by retrieving the current value of C.head and the corresponding block identifier. Next, it attempts to reserve an entry to consume (Line 20), advancing `reserved`. If the reservation succeeds, the consumer reads the data (Line 22) and advances `consumed`. In drop-old mode, the consumer may have to retry consuming if the producers have overwritten the block (Line 24). Reserving an entry can fail in several ways. When the next entry in `blk` is allocated but not yet committed, dequeue returns `BUSY` (Line 26). When `blk` is not fully allocated and all committed entries were already consumed, dequeue returns `EMPTY` (Line 25). Finally, when `blk` is fully committed and fully consumed, the consumer tries advancing C.head (Line 28). Upon success, it retries reserving an entry, jumping to `loop`. Otherwise, dequeue returns `EMPTY`.

Progress guarantees. Similarly to DPDK ringbuffer, BBQ is a deadlock-free queue, its progress depend on a fair scheduler. In contrast to DPDK, BBQ is less affected by CPU oversubscription (see Figure 9g, Section 6). To see why this is the case, consider the situation where DPDK producers form a waiting chain: the last to allocate an entry can only commit once the previous has committed its entry, and so on. This waiting chain hampers the performance because the scheduler is unlikely to unpark the preempted producers in the chain's order. In BBQ, there is no such waiting chain, *i.e.*, producers commit independently. Consumers may wait for producers only in seldom cases. For example, a consumer waits for a preempted producer on the same block if the producer has allocated but not yet committed an entry. Nevertheless, any order in which the fair scheduler unparks preempted producers allows the consumer to make progress.

3.3 Two-Level Control Variables

Essentially, BBQ splits the control variables into two levels, namely the queue-level and block-level variables (see Fig. 2). Queue-level control variables point to blocks (C.head and P.head), whereas block-level control variables to entries (`allocated`, `committed`, `reserved`, and `consumed`).

Versions. As in other queues such as DPDK ringbuffer, control variables have to be versioned to identify multiple reuses of the same memory locations and, in this way, avoid ABA problems[1]. Therefore, queue-level control variables have two fields, an *index* pointing to a block and a *version* identifying how many rounds the whole queue has been reused. Similarly, block-level control variables have an *offset* field pointing to an entry within the block and a *version* field identifying how many times the block has been reused.

Phantom heads. Before producers can allocate entries in a block B, one producer has to reset B's `allocated` cursor as well as advance P.head to point to B. Without making both updates atomic, whichever update executes first may trigger an ABA problem as well. To allow both being updated atomically, we introduce the concept of *phantom head*, which is based on the following observation. The index and version values of P.head can be inferred from the versions of all `allocated` cursors in the queue (as described in Sec. 4.2.2). Similarly, the phantom C.head can be inferred from the versions of all `reserved` cursors. Since the phantom P.head (resp. phantom C.head) is implicitely updated whenever any `allocated` cursor (resp. `reserved` cursor) is updated, we use them instead of queue-level head variables.

[1]Often algorithms try to guarantee operation atomicity by reading from a control variable before and after the operation. If the same value A is read both times, the programmer assumes absence of concurrent updates and hence that the operation was atomic. This assumption breaks if other threads can temporarily change the value to $B \neq A$ and then back to A; algorithms in which this situation can occur are said to suffer from the ABA problem [32].

Cached heads. In principle, phantom heads allow us to eliminate the C.head and P.head variables altogether. Unfortunately, phantom heads are costly: To infer them, one needs to compare the cursors of every block. Instead of eliminating C.head and P.head, we consider them to be *cached heads*, *i.e.*, potentially stale values of the phantom heads. Cached heads only exist for performance reasons; their staleness does not affect correctness.

4 Implementation of BBQ

Figure 4 shows the low-level detail of BBQ, including data-fields, enqueue and dequeue operations for retry-new mode and drop-old mode. The drop-old mode will be introduced in Sec. 4.3.

4.1 Structure

Heads and cursors. `BBQ` has two queue-level `Head` variables and four block-level `Cursor` variables in each block. `Head` and `Cursor` types are 64-bit integers, which can be atomically updated. We reserve two bit-segments in `Head` to represent the version and index and two bit-segments in `Cursor` to represent the version and offset. Given a total number of blocks (`BLOCK_NUM`) and the capacity of a block (`BLOCK_SIZE`), the segments have the following bit-lengths:

$$\begin{aligned} |\texttt{Index}| &= log_2(\texttt{BLOCK\_NUM}) \text{ bits} \\ |\texttt{Offset}| &> log_2(\texttt{BLOCK\_SIZE}) \text{ bits} \\ |\texttt{Version}| &= 64 - max(|\texttt{Index}|, |\texttt{Offset}|) \text{ bits} \end{aligned}$$

The bit-length of `Offset` is larger than $log_2(\texttt{BLOCK\_SIZE})$ to allow for `FAA`-overflow detection. The `Index` and `Offset` are the least significant bits of `Head` and `Cursor`, respectively; `Version` bits immediately follow them; and reminder bits, if existent, are set to 0 and ignored. That allows us to easily manipulate these fields with `FAA` and `MAX` instructions.

For convenience, we access the bit-segments from `Head` and `Cursor` variables as if they were regular fields named `idx`, `off`, and `vsn`, *e.g.*, `allocated.idx`. Moreover, we construct variables (*e.g.*, `Head`) with the short-hand notation `Head{.vsn=version, .idx=index}`, initializing unspecified fields with 0. We may omit the type when clear from the context.

Initially, `idx` and `off` in the first block are zero and for remaining blocks `off` is set to `BLOCK_SIZE`. The initial value of `vsn` will be introduced in Sec. 4.2.2.

Other types. `Block` has shared cursors, annotated with `shared<>`, and an array of entries of type `T` (Line 38). `EntryDesc` is an entry descriptor; it points to a block and contains `offset` to location the actual entry and a `version` data-consistency checks used in drop-old mode (Line 41). Finally, `BBQ` contains the shared heads and an array of `Block<T>`.

```
1  <Head, Block> BBQ<T>::get_phead_and_block(){
2    ph = LOAD(phead);
3    return (ph, blocks[ph.idx]);
4  }
5  state BBQ<T>::allocate_entry(Block blk){
6    if (LOAD(blk.allocated).off >= BLOCK_SIZE)
7      return BLOCK_DONE;
8    old = FAA(blk.allocated, 1).off;
9    if (old >= BLOCK_SIZE)
10     return BLOCK_DONE;
11   return ALLOCATED(EntryDesc{.block=blk, .offset=old});
12 }
13 void BBQ<T>::commit_entry(EntryDesc e, T data){
14   e.block.entries[e.offset] = data;
15   ADD(e.block.committed, 1);
16 }
17 state BBQ<T>::advance_phead(Head ph) {
18   nblk = blocks[(ph.idx + 1) % BLOCK_NUM];
19   cons = LOAD(nblk.consumed);
20   if (cons.vsn < ph.vsn ||
21       (cons.vsn == ph.vsn && cons.off != BLOCK_SIZE)) {
22     reserved = LOAD(nblk.reserved);
23     if (reserved.off == cons.off) return NO_ENTRY;
24     else return NOT_AVAILABLE;
25   }
26   cmtd = LOAD(nblk.committed);
27   if (cmtd.vsn == ph.vsn && cmtd.off != BLOCK_SIZE)
28     return NOT AVAILABLE;
29   MAX(nblk.committed, Cursor{.vsn=ph.vsn + 1});
30   MAX(nblk.allocated, Cursor{.vsn=ph.vsn + 1});
31   MAX(phead, ph + 1);
32   return SUCCESS;
33 }
34 class BBQ<T> {
35   shared<Head> phead, chead;
36   Block<T>[] blocks;
37 }
38 class Block<T> {
39   shared<Cursor> allocated, committed;
40   shared<Cursor> reserved, consumed;
41   T[] entries;
42 }
43 class EntryDesc {
44   Block block; Offset offset; Version version; }
```

```
45 <Head, Block> BBQ<T>::get_chead_and_block(){
46   ch = LOAD(chead);
47   return (ch, blocks[ch.idx]);
48 }
49 state BBQ<T>::reserve_entry(Block blk){
50 again:
51   reserved = LOAD(blk.reserved);
52   if (reserved.off < BLOCK_SIZE) {
53     committed = LOAD(blk.committed);
54     if (reserved.off == committed.off)
55       return NO_ENTRY;
56     if (committed.off != BLOCK_SIZE){
57       allocated = LOAD(blk.allocated);
58       if (allocated.off != committed.off)
59         return NOT_AVAILABLE;
60     }
61     if (MAX(blk.reserved, reserved + 1) == reserved)
62       return RESERVED((EntryDesc){.block=blk,
63         .offset=reserved.off, .version=reserved.vsn});
64     else goto again;
65   }
66   return BLOCK_DONE(reserved.vsn);
67 }
68 T BBQ<T>::consume_entry(EntryDesc e){
69   data = e.block.entries[e.offset];
70   ADD(e.block.consumed, 1);
71   allocated = LOAD(e.block.allocated);
72   if (allocated.vsn != e.version) return NULL;
73   return data;
74 }
75 bool BBQ<T>::advance_chead(Head ch, Version vsn){
76   nblk = blocks[(ch.idx + 1) % BLOCK_NUM];
77   committed = LOAD(nblk.committed);
78   if (committed.vsn != ch.vsn + 1)
79     return false;
80   MAX(nblk.consumed, Cursor{.vsn=ch.vsn + 1});
81   MAX(nblk.reserved, Cursor{.vsn=ch.vsn + 1});
82   if (committed.vsn < vsn + (ch.idx == 0))
83     return false;
84   MAX(nblk.reserved, Cursor{.vsn=committed.vsn});
85   MAX(chead, ch + 1);
86   return true;
87 }
```

retry-new mode	drop-old mode

Figure 4: Low-level details of BBQ.

4.2 Operations

Enqueue and dequeue operations are divided into different cases: First, when the allocation in the enqueue or the reservation in the dequeue do not fail. Second, when enqueue or dequeue have to advance respective heads to the next block.

4.2.1 Successful allocation/reservation

The producer uses FAA to allocate an entry (Line 8) and returns its location as `EntryDesc` if there is enough space in the current block (Line 11). A pre-check (Line 6) avoids endless increasing of `allocated` when the queue is full, which could cause FAA overflows and impact performance negatively. For the consumer, the entry is reserved through MAX[2] (Line 61), which atomically sets a variable if the given value is greater than the variable's value and returns the old value. Consumers never pass producers (Line 54) and can read when out-of-order commit are not ongoing in the same block, which means all allocated entries are committed (Line 58).

4.2.2 Advancing to the next block

Monotonic version updates. Head and cursor versions are initially zero. Both enqueue and dequeue calls start by reading the current cached head (`phead` and `chead`, respectively) into a local variable (`ph` and `ch` in Fig. 3). After failing to allocate or reserve an entry, these calls try to advance the respective phantom heads by calling advance_phead or advance_chead.

[2]Unlike FAA, MAX provides conditional update semantics. Moreover, for some cases, MAX has similar semantics to CAS but better performance observed from experimental results. We use CAS and while loop to achieve the same functionality for architectures that do not support MAX such as x86 [16].

These functions *try* to reset the cursors of the next block with the previously read version of the cached head plus one (Lines 29, 30, 80, and 81 in Fig. 4). Subsequently, the functions *try* update the cached head itself (Lines 31 and 85).

The reset of cursors and the update of cached head may not always succeed. Consider the following example. Two producers try to allocate entries at block B_0 and fail. Both have read `phead` with value `{.vsn=0,.idx=0}`. Now both call advance_phead concurrently and are at Line 30. Producer P_1 stalls while producer P_2 succeeds updating the `allocated` cursor of block B_1. P_2 also allocates one or more entries such that now B_1's `allocated` has the value `{.vsn=1,.off=16}`. If now P_1 would be able to succeed resetting `allocated`, then the allocations of P_2 would be lost. Nevertheless, to avoid such ABA situations, the reset of cursors and update of cached head do not have to be performed with an expensive `CAS`. The recent `MAX` atomic instruction from Armv8.1-LSE can provide the required monotonicity.

Invariants. Producers have to ensure they advance `phead` only if the next block that has no unconsumed data. Consumers have to ensure they advance `chead` only if the next block has committed data.

We guarantee these invariants by ensuring that the version difference between phantom `phead` and phantom `chead` never exceeds 1. When producers advance `phead` and reset the `allocated` and `committed` cursors of the next block with version `ph.vsn+1` (Line 30), the `consumed` cursor must have version `ph.vsn` (Line 21). Similarly, when consumers advance `chead` and reset the `reserved` and `consumed` cursors of the next block with version `ch.vsn+1` (Line 80), the `committed` cursor must have version `ch.vsn+1` (Line 79).

Order matters. Often the order in which shared variables are accessed is crucial for correctness. For example, reading `reserved`, `committed`, and `allocated` variables (Lines 51, 53, and 57) in a different order can cause the consumer to read garbage. Moreover, updating cached heads (Lines 31 and 85) must happen after updating block-level variables (Lines 29, 30, 80, and 81), otherwise blocks may be fully skipped.

To guarantee shared variables are accessed in the program order of Fig. 4 on architectures with weak memory models, C/C++ implementations of BBQ can employ atomic `LOAD`, `STORE`, `MAX`, `FAA`, and `CAS` instructions with sequentially consistent memory barriers (see C11/C++11 atomics [6]). In Sec. 5, we report a correct relaxation of these barriers.

4.3 Drop-old Mode

Unlike the retry-new mode, where producers cannot insert data when the queue is full, in drop-old mode, producers continue to write even if the data is not yet consumed. Consequently, producers no longer depend on the consumers to make progress. The FIFO property still holds, except that some data might be lost. In other words, entries are consumed in the order in which they were allocated, but some committed entries may not be consumed.

Speculative reads. Drop-old mode is widely used in profiling scenarios, where enqueue calls writing a log should not be delayed by dequeue calls that read the log. To reduce the chances of dequeue calls interfering with enqueue calls, consumers read data in a speculative fashion. The consumer first reads the data and then checks whether it has been overwritten. If so, it discards the data and tries reserving another entry.

From retry-new to drop-old mode. A few differences exist between retry-new and drop-old mode. First, producers avoid advancing to blocks that are still not fully committed in the previous round, returning `BUSY` (Lines 27 and 28).

Second, consumers guarantee FIFO order by checking if the version of the next block is greater than or equal to the current one (Line 82). If that is the case, `reserved` is reset with the version of `committed` (Line 84), indicating the block is ready to be read. The first block is a special case because, in contrast to other blocks, its version is always off-by-one. Therefore, we add 1 to the comparison if `ch.idx == 0`.

Third, the data-consistency check is based on the fact that a block is not overwritten as long as `allocated` and `reserved` versions are equal. Therefore, before reading data, we record the `reserved` version (Line 63), and after copying the data from the entry, we check if corresponding `allocated` version still matches the `reserved` (Line 72).

4.4 Variable-sized Entries

BBQ can support variable-sized entries with minor algorithmic changes. Each entry has an additional metadata `size` to support different entry sizes in one queue. Block-local cursors and `BLOCK_SIZE` indicate the space of entries instead of their number. `MAX` at Line 61 is no longer sufficient; `CAS` must be used instead.

Dummy entry. Unlike the fixed-size version of BBQ, where entries can exactly fill up a block, here, the remaining space of a block might not be enough to contain the new entry. In such cases, we mark the space with a dummy entry and return `BLOCK_DONE` to trigger a retry in the next block. Since enqueue uses `FAA`, the producers that cause `allocated` go over the boundary marks the dummy entry by setting its `size` to zero and commits it. Consumers that read an entry with size zero ignore the dummy entry and retry in the next block. Upon reading the dummy entry, the consumer also sets `consumed` to be equal to `BLOCK_SIZE`.

4.5 Other Implementation Details

We have implemented BBQ in C and Java. We have also implemented a wrapper with the Java Native Interface (JNI) [20] to call the C version from Java.

Finally, we have optimized BBQ for SPSC scenarios as follows: (1) `phead` and `chead` are no longer shared variables and

can be accessed with non-atomic loads/stores. (2) `allocated` and `committed` (resp. `reserved` and `consumers`) are merged into one variable and updated with `STORE`.

5 Verification and Optimization of BBQ

Concurrent data structures are complicated beasts and are easy to get wrong [30]. To increase confidence in our C implementation and find intricate bugs, we generate a series of small hand-crafted tests that can trigger corner cases in the algorithm and then use VSYNC [42], an extension to the GenMC model checker [36]. The tool generates all executions of the algorithm on those tests, including executions that can only happen on WMMs, exercising the following critical corner cases: (1) queue full or empty, (2) FIFO, (3) wrap-around, and (4) termination of bounded loops with bounded effect [37,42].

Bugs. We found three concurrency bugs in an earlier version of the drop-old mode of BBQ.

1. A test revealed a bug in which enqueue operations incorrectly returned `BUSY`. The block was detected as `NOT_AVAILABLE` because, in that version, the condition at Line 27 was `committed.off == BLOCK_SIZE && committed.vsn == ph.vsn`. Therefore, even if other producers reset the next block and have the space to allocate, the block would still be `NOT_AVAILABLE`. That violated linearizability.
2. We have found a termination bug in which the checking in Line 82 was written as `blk.committed.vsn >= nblk.committed.vsn`, missing the special case of the version number in the first block, which may let consumers advancing the block forever if the queue is empty and all blocks happen to have the same version number.
3. The wrap-around test revealed a bug due to a missing fence, where readers could return incorrect data when a fast writer overwrote the entry they were currently reading.

We found these bugs through the verification with model checking. They were not found during stress testing, nor by running the small test cases directly on hardware. However, we could retrospectively construct test cases that reproduce these bugs on real hardware. Concurrent algorithms, especially those using complicated synchronization such as drop-old mode, are hard to get right using traditional methods.

Barrier optimization. We used VSYNC to run the memory barrier optimization for WMMs. The results consistent with the order analyze of reading/updating shared variables in Sec. 4.2.2. For the fixed entry size version, 14 atomic instructions with full memory barriers are optimized to 3 release barriers, 3 acquire barriers, and 8 relaxed barriers, respectively.

6 Evaluation

6.1 Environment Setup

Hardware. All of our experiments are performed on three x86 machines with 88, 96, and 12 hyperthreads, respectively (denoted as x86-88T, x86-96T, and x86-12T), and an ARM machine with 96 cores (arm-96T). x86-88T and x86-96T are connected through two 10Gbps links.

Software. On these servers, we installed Ubuntu 20.04.3 LTS, with Linux kernel 5.4.0. We use Linux perf [26] to get results of L1 cache misses, the version of it is the same with the Linux kernel. Java-based experiments use JDK v11 [19].

6.2 Microbenchmarks

Workloads. We have the following 3 workloads for microbenchmarks implemented in C/C++ and Java:

- simple: Each producer or consumer has its own thread, where they keep executing enqueue or dequeue operations in a loop. Data is validated after each dequeue.
- complex: Based on the simple workload. Producers and consumers allocate space for data, preform enqueue and dequeue then manually free (C/C++ version) or let JVM garbage collection it [46] (Java version). Additionally, each operation also performs a deterministic random busy-loop of at most one hundred `nop` instructions.
- profiling: Based on the simple workload. The throughput of producers and consumers is fixed at 10*k*op/s and 1*k*op/s, respectively.

Thread affinity. For MPSC or SPMC scenarios, we assign a single producer or consumer at the first core/hyperthread and then distribute the other threads sequentially to cores/hyperthreads. For MPMC, we assign producers and consumers interleaved one by one; if their number differs, the surplus is assigned at the end.

Experiments. We perform the following experiments, each measuring a different metric:

- throughput: Total number of consumed entries per second.
- data-latency: Average time each data stays in the queue.
- op-latency: Average latency of each enqueue or dequeue operation.
- cache-miss: Average number of L1 cache misses per consumed entry, measured with Linux `perf`.
- fairness: Throughput of each producer and consumer (only for MPSC and SPMC).
- full/empty: Latency of enqueue when the queue becomes full and latency of dequeue when the queue becomes empty (only used with simple workload).
- oversubscription: Throughput with more producers and consumers than than cores/hyperthreads.

Each experiment runs 3 times. If not specified otherwise, solid lines represent average results; shaded area represents

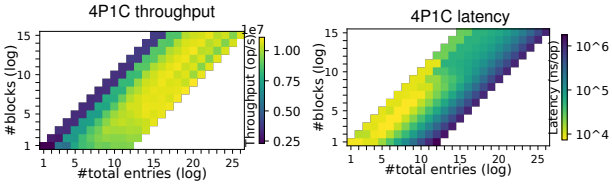


Figure 5: BBQ throughput and latency varying number of blocks and entries (x86-88T).

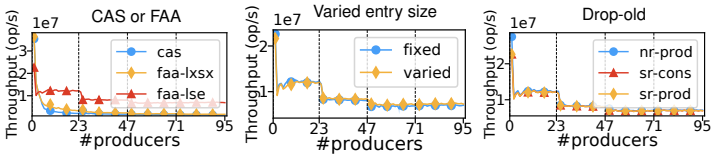


Figure 6: BBQ throughput with CAS and FAA; with support for variable-sized entries; and with drop-old mode (arm-96T).

standard deviation; and vertical dashed lines indicate when threads cross NUMA nodes, are assigned to hyperthreads in the same core, or are oversubscribed.

Configuration. The data size is always 8 bytes, a size all queues can support. For the data-latency experiment, the number of entries is around 128. For the other experiments, the buffer size is 32*k* bytes unless specified otherwise.

6.2.1 BBQ Parameters and Design Choices

We start by evaluating parameters and design choices of BBQ.

Configuring the number of blocks. Figure 5 shows throughput and data-latency experiments for BBQ with four producers and one consumer. The color scale shows the existing trade-off between number of entries and number of blocks; users have to be aware of that when choosing the buffer size and number of blocks. We use the following heuristic function to determine the number of blocks in all rest experiments: $\textit{number of blocks (log)} = \max(1, \lfloor \textit{number of entries (log)}/4 \rfloor)$.

Performance impact of FAA. Figure 6 shows the results of an MPSC throughput experiment on our Arm machine. BBQ is configured to use FAA instruction from Armv8.1 LSE, standard FAA and CAS implemented with load-exclusive and store-exclusive instructions. Except for the 1 thread case, LSE-based FAA shows the best scalability, outperforming the other two by at least 5 times.

Support for variable-sized entries. Figure 6 also shows the throughput of the BBQ with fixed- and variable-sized entries. The size of each data is the same, yet the varied entry version has to store additional size information for every entry. Nevertheless, the throughput difference between both is negligible.

Consumer-producer interference in drop-old mode. Finally, Figure 6 shows the throughput of BBQ with drop-old mode in two configurations: MPSC and MPNC (multiple-producer/no-consumer). The throughput of the producers (nr-prod) with no consumers is less than 8% higher than with consumer (sr-prod). Moreover, the consumers manage to consume at least 99.97% of the entries except for the 1 thread case (sr-cons). These results illustrate that consumers with the speculative-read method incur a rather minor interference on producers — please refer to Sec. 6.2.3 for a baseline with existing implementations.

6.2.2 State-of-the-art Comparison: Retry-new Mode

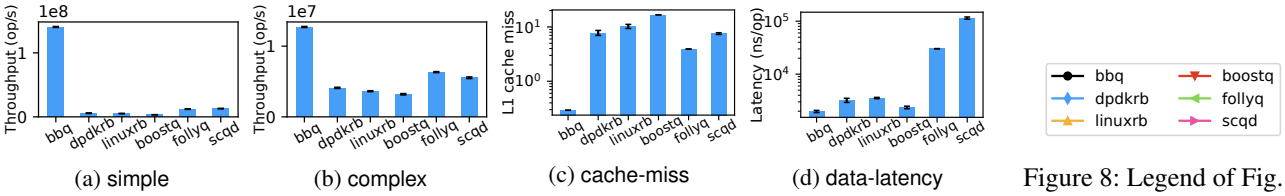
We now compare BBQ against 5 state-of-the-art bounded queues: dpdkrb, DPDK ringbuffer v21.08 [9]; scqd, a lock-free bounded queue [40]; linuxrb, the ringbuffer in the Linux kernel v5.16 [22]; boostq, the bounded queue in C++ Boost libraries v1.71 [4]; and follyq, the bounded queue (with total method) [25] in Meta's open-source Folly library v2021.11.8. For dpdkrb and follyq, we use their SPSC versions to run corresponding SPSC experiments.

Effectiveness of the Block-based Approach. To isolate the effect of the blocks, we first focus on SPSC experiments because BBQ do not profit from FAA in such scenarios. Figure 7 shows that BBQ greatly outperforms all other bounded queues in all experiments. For the simple workload, BBQ yields 11.3x to 42.4x higher throughput than other libraries. The throughput of BBQ is $1.41 \cdot 10^8$ op/s, while the second-best one follyq is $1.24 \cdot 10^7$ op/s. For the complex workload, which has a random busy-loop to limit the maximum throughput, BBQ still outperforms follyq by 2x. BBQ's better performance is mainly due to the massive decrease in L1 cache misses with the block-based approach (notice the y-axis log scale).

Throughput in MPSC and SPMC scenarios. Figures 9a and 9b show BBQ performing on par or better than other queues in the simple and complex workloads. For MPSC scenarios, BBQ performs up to 10.13x and 3.65x faster than the second-best queue, respectively. For SPMC scenarios, BBQ performs up to 1.88x and 2.39x faster than the second-best queue, respectively.

The throughput difference between MPSC and SPMC results can be attributed in part to the different L1 cache misses measurements (see Fig. 9c). BBQ consumers employ CAS operations in every dequeue, and these can fail and have to be retried, each time suffering another cache miss.

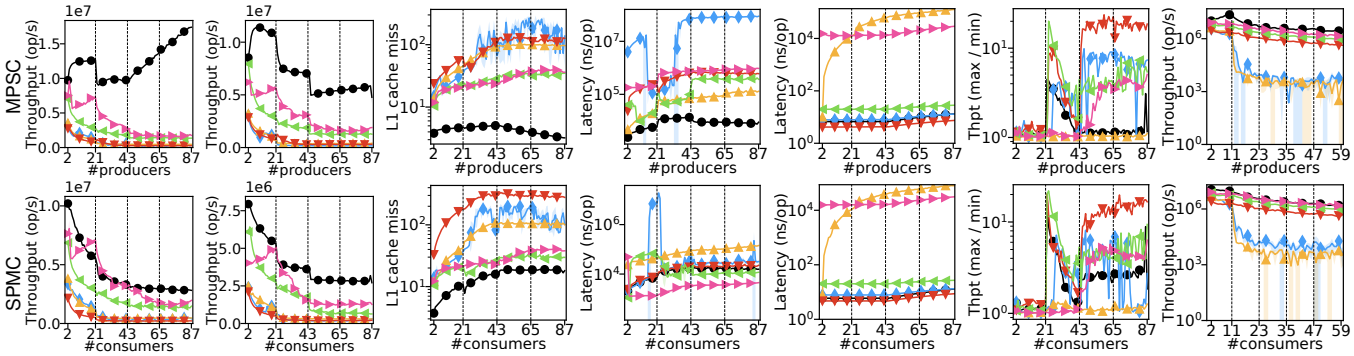
Data latency. We measure the average time data stays in the queue in the complex workload, as shown in Fig. 9d. For MPSC case, BBQ performs consistently better than other bounded queues; up to 17.22x lower latency than the second-best queue. For the SPMC case, scqd performs best, up to 7.45x lower latency than BBQ. That is an artifact of the delayed invalidation trick (see Sec. 2): Once the queue is empty (C.head = P.tail), consumers invalidate the entries pointed by C.head after a delay. Since consumers first increment C.head and then wait, multiple consumers will be pending on differ-



(a) simple (b) complex (c) cache-miss (d) data-latency

Figure 8: Legend of Fig. 9

Figure 7: SPSC comparison of BBQ against state-of-the-art on x86-88T.



(a) simple (b) complex (c) cache-miss (d) data-latency (e) full/empty (f) fairness (g) oversubscription

Figure 9: MPSC and SPMC comparison of BBQ against state-of-the-art on x86-88T (and x86-12T for oversubscription).

ent entries. As soon as the producer commits a new entry, one consumer aborts its delay and immediately returns the data.

Full and empty queues. Figure 9e shows the latency for failed enqueue on a full queue (top figure), and failed dequeue on an empty queue (bottom figure). In such scenarios, the delay invalidation of scqd incurs a high cost: the latency of failed operations in scqd is around 1000x higher than in most other queues. For linuxrb, the latency increases with the number of producers/consumers due to its coarse-grained locking.

Fairness between producers or consumers. Figure 9f shows the relation between maximum and minimum throughput of producers (top figure) and consumers (bottom figure). linuxrb provides exceptional fairness because it relies on a fair spinlock[3]. Other queues show unfair throughput after crossing the first NUMA node (at 22 producers/consumers) except for scqd, which becomes unfair when hyperthreads of the same cores start being used (at 44 producers/consumers).

Oversubscription effects. Figure 9g shows the results of our oversubscription experiment on x86-12T with up to 5x more threads than hyperthreads. Both dpdkrb and linuxrb are highly affected by oversubscription; the former due to their in-order policy (see Sec. 2), the latter due to its coarse-grained locking. Under oversubscription (*i.e.*, with more than 12 threads), BBQ outperforms the second-best queue by a small margin: 2.22x in MPSC and 1.23x in SPMC scenarios.

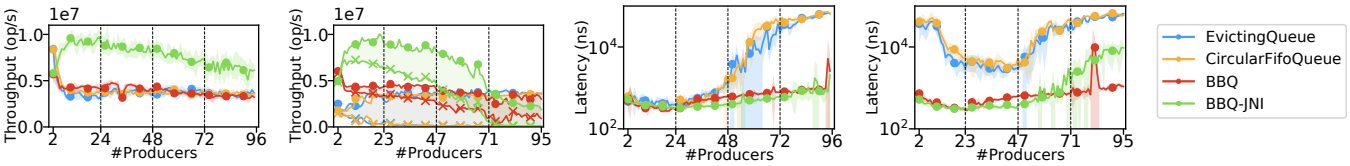
[3]In our userspace port of linuxrb, we employ a ticket lock.

6.2.3 State-of-the-art Comparison: Drop-old Mode

We now compare BBQ with other two bounded queues that support overwriting old values, namely EvictingQueue from Google Core Libraries Guava [15], and CircularFifoQueue from Apache Commons [1]. The experiments are conducted on the arm-96T machine.

Producer performance. Figure 10a shows the enqueue throughput with *no consumers* for the complex workload. On the one hand, BBQ-JNI yields 3.2x higher enqueue throughput than EvictingQueue and CircularFifoQueue. On the other hand, BBQ yields an enqueue throughput rather similar to them. Intuitively, BBQ-JNI has a better performance since employs real `FAA` instructions, whereas, in the Java version of BBQ, the JVM translates `FAA` into `CAS` [35].

Figure 10c shows the enqueue latency, again with *no consumers*, for the profiling workload. Remember that producers issue 10k enqueue calls per second in the profile workload. With BBQ and BBQ-JNI, the enqueue latency slowly increases: 147.9ns and 176.4ns with 1 thread, respectively, to 965.6ns and 914.3ns with 94 threads, respectively. Up to 44 producers, EvictingQueue and CircularFifoQueue perform similar to BBQ variants. With more than 44 producers, however, their enqueue latency quickly increases up to 70 μs (72x higher than BBQ). From Fig. 10a, we know that their maximum enqueue throughput is about 450$k$op/s. Hence, these queues already reached throughput limit with 44 producers, and any additional producers can only increase the latency. We believe the spike at 95 threads (BBQ with 5.1μs and BBQ-JNI with 2.0μs)



(a) MPNC complex (b) MPSC complex (c) MPNC profiling (d) MPSC profiling

Figure 10: Cross comparison results for drop old mode on arm-96T.

may caused by garbage collection, but further investigation is necessary.

Enq-deq interference. We now introduce a single consumer to understand the interference of dequeue on the enqueue operations. Ideally, the enqueue operations should incur a small overhead (latency) to the profiled program; and this overhead should be minimally affected by concurrent dequeue calls. Moreover, if enqueue calls interfere with dequeue calls too frequently, more data may be dropped, *i.e.*, overwritten before being consumed.

Figure 10b shows the enqueue and dequeue throughput (marked with • and ×, respectively) for the complex workload. Comparing Figs. 10a and 10b, we observe that the enqueue throughput of BBQ is similar in both figures and of BBQ-JNI is similar up to 47 threads, but after that it drops to about $1.89 \cdot 10^6$ op/s. The enqueue throughput of EvictingQueue and CircularFifoQueue is initially lower when a consumer is concurrently calling dequeue. The reason for this lower enqueue throughput can explained by observing the difference between enqueue and dequeue in Fig. 10b.

First, note that with more than a few producers, the enqueue and dequeue throughput of each queue do not match, *i.e.*, the consumer is not fast enough to read out all the data before the producers start overwriting the oldest entries. Also note that the more the enqueue throughput of EvictingQueue and CircularFifoQueue recovers (by increasing producers), the lower is the dequeue throughput. Once their throughput is back to the level of Fig. 10a with 15 threads, their dequeue throughput is no more than $4.92 \cdot 10^5$ op/s. In contrast, BBQ and BBQ-JNI sustain a much higher dequeue throughput up to 46 threads ($2.89 \cdot 10^6$ op/s and $5.07 \cdot 10^6$ op/s, respectively).

Figure 10d shows the enqueue latency for the profiling workload. BBQ and BBQ-JNI provide enqueue latencies varying from 730.6ns and 519.9ns with 2 producers, respectively, up to 1082ns and 9503ns with 95 producers — we ignore the noisy region with 81 producers. Comparing the results of Figure 10c and Figure 10d reveal that the enqueue latency of EvictingQueue and CircularFifoQueue, for example with 8 producers increase by 124.97 times when adding a single consumer with a relatively low dequeue frequency.

The latency increases as well as the throughput decreases of BBQ-JNI after 47 producers could be related to the JNI overhead of calling C code from Java.

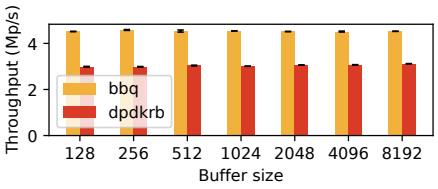


Figure 11: Throughput comparison between BBQ or DPDK ring buffer.

6.3 Macrobenchmarks

We now explore three benchmarks that represent the real-world usage queues.

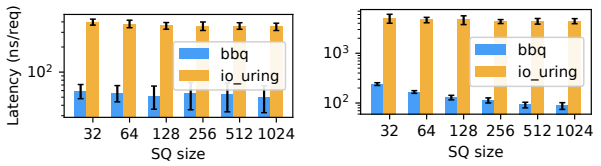
6.3.1 DPDK's End-to-end Benchmark

We replace the ring buffer in DPDK's event library [12] and network driver [11] with BBQ, and run the multiprocess benchmark [8] from the DPDK Test Suite [10] (DTS). The benchmarks consists of one server process receiving and distributing packets, and two client processes performing level-2 packet forwarding [7]. These processes run on the device under test (DUT), our x86-88T machine. The tester and traffic generator TRex [29] run on our x86-96T machine. The packet size is 64 bytes (along with the UDP header) as well as the entry size of the queue. The versions of DPDK, DTS, and TRex are 21.08, 21.02, and 2.92, respectively. We report the end-to-end throughput (in million packets per second) measured by the traffic generator.

Figure 11 shows our experimental results. BBQ provides 1.5x higher throughput with different buffer sizes in the driver. We observed no further improvements with larger buffer sizes, indicating that the ring buffer may not be a bottleneck any more. We also replaced the so-called software queue in the multiprocess benchmark, and observed no improvement.

6.3.2 Linux io_uring

Linux io_uring [13] is a new asynchronous I/O [31] API for kernel-user space communication. It consists of two ring buffers, one for request submissions (SQ) and another for completion confirmations (CQ). It supports batched submission and batched confirmations with configurable batch sizes [18].



(a) Batchsize = 1 (b) Batchsize ∈ [1, 32]

Figure 12: Latency per request comparison of BBQ and Linux io_uring on x86-88T.

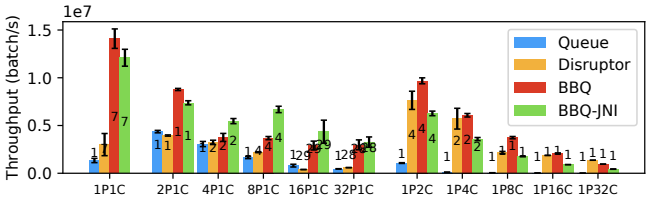


Figure 13: Throughput comparison of BBQ and BBQ-JNI against LMAX Disruptor on x86-88T.

We port io_uring from Linux kernel (v5.14-rc6) [17] to userspace, omitting I/O related functionality and replacing its ring buffers with BBQ. To avoid unstable results, we disable the option of overflowing entries into an additional linked list. We set the CQ size to twice the SQ size as recommended [21]. Our benchmark runs three threads: The first submits request batches (via SQ); the second (representing the kernel) consumes them and immediately produces confirmations (via CQ); and the third consumes the confirmation batches. We configure submission and confirmation batches with size 1 or with a random value from 1 to 32. Each experiment runs 10 times, measuring the time to submit 1M requests.

Figure 12 shows a significant improvement of the latency per request when using BBQ. For example, with batch size of 1 and SQ size of 32 and 1024, BBQ yields 6.7x and 6.9x lower latency than the original ring buffer, respectively. For random batch size and the same SQ sizes, BBQ yields even lower latencies: 20.9x and 50.5x, respectively.

6.3.3 LMAX Disruptor Benchmarks

Disruptor [23] is concurrency mechanism used for high-performance financial exchange. Its core component is a ring buffer. We compare its throughput with the Java and JNI versions of BBQ with three official Disruptor benchmarks: `OneToOneThroughputTest`, `ThreeToOneThroughputTest`, and `OneToThreeThroughputTest`. We modify these benchmarks to support not just three, but more producers or consumers. Apart from this modification, all other parameters (*e.g.*, number of iterations, sleep time between operations, number of repetitions) are unchanged.

Disruptor can randomly change the batch size based on the workload. To make the comparison as fair as possible, we first run the benchmark with Disruptor to get the average batch size used, and then run BBQ with that batch size. Figure 13 shows the throughput of Disruptor, BBQ, and the baseline Java queue (`java.util.concurrent.BlockingQueue`) for several scenarios. The number on each bar refers to the (average) batch size, and the label *p*P*c*C indicates the number of producers (*p*) and consumers (*c*).

In the 1P1C scenario, Disruptor yields almost 3x higher throughput than the Java queue (3 Mop/s versus 1.3 Mop/s). BBQ and BBQ-JNI, however, yield an order of magnitude higher throughput (14.1 Mop/s and 12.1 Mop/s, respectively). The higher performance of BBQ over BBQ-JNI is due to the JNI call overheads. With 8 producers, the difference between Disruptor and BBQ is lower (2.2 Mop/s and 3.7 Mop/s, respectively). BBQ-JNI yields 3x Disruptor's throughput (6.7 Mop/s) due to its use of `FAA`. With 32 producers, BBQ and BBQ-JNI again outperform Disruptor by an order of magnitude (3.0 Mop/s, 3.3 Mop/s, and 0.6 Mop/s, respectively).

With a single producer and multiple consumers, BBQ-JNI has no opportunity to gain performance by using `FAA`. Due to that, its performance pays the penalty of the JNI call overheads. Nevertheless, BBQ still outperforms Disruptor in most configurations. For example, BBQ yields 1.23x higher throughput than Disruptor with 2 consumers (1P2C); and 1.68x higher throughput with 8 consumers. With 32 consumers, Disruptor yields 1.42x higher throughput than BBQ.

7 Conclusion

We presented BBQ, a novel ringbuffer design that dramatically reduces the enq-deq interference by splitting the entire ringbuffer into multiple blocks. BBQ is applicable to a large spectrum of scenarios, from exchanging data to profiling, with single or multiple producers/consumers, sending fixed- or variable-sized entries, among others. Our experimental results show that BBQ outperforms several industrial ringbuffers (*e.g.*, DPDK, LMAX Disruptor, Linux io_uring, Meta's Folly queue) in the great majority of workloads.

To support modern architectures such as Armv8.1, we verified and optimized BBQ with a model checker for weak memory models. Even though far from sound, verification with model checkers has proven a valuable, low-cost method of catching bugs.

Currently, our io_uring benchmark evaluates whether BBQ is promising for such scenarios without involving kernel details. In the future, we plan to port BBQ to kernel space to replace Linux io_uring.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments. We specially thank Bohdan Trach for the helpful discussions and for proofreading this manuscript.

References

[1] Apache Commons. http://commons.apache.org/.

[2] Arm A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0596/2021-09.

[3] Arm architecture reference manual armv8, for a-profile architecture. https://developer.arm.com/documentation/ddi0553/latest.

[4] Boost C++ Libraries. https://www.boost.org/.

[5] BPF ring buffer. https://www.kernel.org/doc/html/latest/bpf/ringbuf.html.

[6] C++ Atomic operations library. https://en.cppreference.com/w/cpp/atomic/atomic.

[7] Cisco Layer Two Forwarding (Protocol) "L2F". https://datatracker.ietf.org/doc/html/rfc2341.

[8] Client-Server Multi-process Example. https://doc.dpdk.org/guides/sample_app_ug/multi_process.html.

[9] Data Plane Development Kit. https://www.dpdk.org/.

[10] Data Plane Development Kit Test Suite. https://doc.dpdk.org/dts/gsg/.

[11] dpdk/drivers/net/ring. https://github.com/DPDK/dpdk/tree/main/drivers/net/ring.

[12] dpdk/lib/eventdev. https://github.com/DPDK/dpdk/tree/main/lib/eventdev.

[13] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[14] Folly: Facebook Open-source Library. https://github.com/facebook/folly.

[15] Guava: Google Core Libraries for Java. https://github.com/google/guava.

[16] Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[17] io_uring source code. https://elixir.bootlin.com/linux/v5.14-rc6/source/fs/io_uring.c.

[18] io_uring_enter - initiate and/or complete asynchronous I/O. https://unixism.net/loti/ref-iouring/io_uring_enter.html.

[19] Java Development Kit. https://jdk.java.net/.

[20] Java Native Interface Specification. https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html.

[21] liburing. https://github.com/axboe/liburing.

[22] Linux Kernel Circular Buffers. https://www.kernel.org/doc/html/latest/core-api/circular-buffers.html.

[23] LMAX Disruptor: A High Performance Inter-Thread Messaging Library. https://github.com/LMAX-Exchange/disruptor.

[24] Lockless Ring Buffer Design. https://www.kernel.org/doc/Documentation/trace/ring-buffer-design.txt.

[25] MPMC Queue. https://github.com/facebook/folly/blob/main/folly/MPMCQueue.h.

[26] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[27] Producer/consumer synchronization modes. https://doc.dpdk.org/guides/prog_guide/ring_lib.html#producer-consumer-synchronization-modes.

[28] RISC-V. https://riscv.org/.

[29] TRex: Realistic Traffic Generator. https://trex-tgn.cisco.com/.

[30] Unread entries potentially lost in buf_ring after ABA condition. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=246475.

[31] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[32] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192. IEEE, 2010.

[33] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, USA, 2011.

[34] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[35] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic instructions in java. In *European Conference on Object-Oriented Programming*, pages 133–154. Springer, 2002.

[36] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 96–110, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. Making weak memory models fair. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[38] 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*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. Association for Computing Machinery.

[39] Ruslan Nikolaev. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue . https://github.com/rusnikola/lfqueue.

[40] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[41] Ruslan Nikolaev and Binoy Ravindran. Wcq: A fast wait-free queue with bounded memory usage. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 461–462, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 530–545, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Or Ostrovsky and Adam Morrison. Scaling concurrent queues by using htm to profit from failed atomic operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 89–101, 2020.

[44] Nicholas A Solter and Scott J Kleper. *Professional C++*. John Wiley & Sons, 2005.

[45] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 286–297, 2017.

[46] Bill Venners. The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*, 1998.

[47] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.