

Durable Queues: The Second Amendment

Gal Sela
galy@cs.technion.ac.il
Technion
Israel

Erez Petrank
erez@cs.technion.ac.il
Technion
Israel

ABSTRACT

We consider durable data structures for non-volatile main memory, such as the new Intel Optane memory architecture. Substantial recent work has concentrated on making concurrent data structures durable with low overhead, by adding a minimal number of blocking persist operations (i.e., flushes and fences). In this work we show that focusing on minimizing the number of persist instructions is important, but not enough. We show that access to flushed content is of high cost due to cache invalidation in current architectures. Given this finding, we present a design of the queue data structure that properly takes care of minimizing blocking persist operations as well as minimizing access to flushed content. The proposed design outperforms state-of-the-art durable queues.

We start by providing a durable version of the Michael Scott queue (MSQ). We amend MSQ by adding a minimal number of persist instructions, fewer than in available durable queues, and meeting the theoretical lower bound on the number of blocking persist operations. We then proceed with a second amendment to this design, that eliminates accesses to flushed data. Evaluation shows that the second amendment yields substantial performance improvement, outperforming the state of the art and demonstrating the importance of reduced accesses to flushed content. The presented queues are durably linearizable and lock-free. Finally, we discuss the theoretical optimal number of accesses to flushed content.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms; Concurrent algorithms; Theory of computation** → **Data structures design and analysis; Hardware** → *Emerging architectures; Non-volatile memory.*

KEYWORDS

Non-Volatile Memory; Concurrent algorithms; Concurrent Data Structures; Durable Linearizability; Lock-Freedom; FIFO Queue

The conference version of this paper is available at [46], and the code is publicly available at <https://github.com/galysela/DurableQueues>.

1 INTRODUCTION

Byte-addressable non-volatile memory combines DRAM’s byte-accessibility, with the durability and size of storage. Various technologies, such as resistive random access memory [2], phase-change memory [41] and 3D XPoint [25], are expected to become available soon, with Intel/Micron 3D XPoint already available to consumers (under the brand name Optane). Non-volatile RAM (abbreviated as NVRAM) is expected to co-exist or replace DRAM in upcoming architectures, allowing program’s modifications to its data structures

survive system crashes. NVRAM platforms are expected to make a fundamental change in the design of the computing infrastructure including file systems, databases and other computations that process persistent data.

While data stored in main memory will survive a crash, without further technological development, the caches and machine registers remain volatile, losing their content during a crash. This creates a consistency challenge, because writes may not reach the memory at the time and order the processor issues them. When programs write data to memory, the CPU does not access the memory directly, but rather writes to the cache and the data only later gets flushed back to memory. Furthermore, the order in which cache lines get written back to the memory is unpredictable, as cache line evictions are triggered by local needs to make room for new cache content. This process may cause the state of the memory after a crash to become inconsistent, reflecting some modifications but missing others, impeding correct recovery.

In order to make sure that the memory contains the required data for a potential crash and recovery, special instructions are used to force the flushing of cache lines from the cache to the memory. Asynchronous flush instructions initiate a cache line flush and let other instructions proceed while the data is being copied to memory. An additional synchronous fence (such as Intel’s SFENCE instruction) makes sure that the flushing becomes visible before any other memory instruction becomes visible to other threads. The fence instruction is blocking and costly and therefore durable algorithms have attempted to reduce the use of SFENCE to achieve better performance. Cohen et al. [9] have shown that a durably linearizable [27] lock-free [21] object must use at least one fence instruction per update operation at worst case. They also presented a universal construction that achieves this bound, but their universal construction was intended as a proof of existence and no attempt was made to provide acceptable performance.

The initial goal of this project was to optimize the performance of a durable FIFO queue. FIFO queues are used at the core of several existing persistent messaging systems (e.g., IBM MQ [24], Oracle Tuxedo MQ [37], Rabbit MQ [48] and many more). Currently these queues are structured to suit the block-based interface of HDDs and SSDs. This design incurs costs like marshaling queue updates in streams, file system calls to persist message queues, etc., and so an adaptation to NVRAM platforms can bring a dramatic improvement to the queues performance and future use.

Following previous work in this area, we focused on reducing the number of blocking persist operations. We started with the lock-free queue of Michael and Scott [35] (denoted henceforth MSQ), which was used in previous work [16] due to its wide applicability to all architectures. We amended MSQ in two different manners, obtaining two novel durably linearizable lock-free queue constructions with a minimal number of blocking persist operations: one blocking

This work was supported by the United States - Israel BSF grant No. 2018655.

persist operation for any data structure modification operation. This meets the lower bound of Cohen et al. [9]. These two optimal durable queue algorithms are the first contribution of this paper.

One of these two algorithms, called `UnLinkedQ`, is designed in the spirit of [57] to avoid persisting the underlying node links. In this algorithm, we allocate the nodes on designated areas, in which the recovery procedure can look for valid nodes of the queue. This requires a new persistent ordering mechanism that allows the recovery to determine the order of nodes in the queue without incurring a large overhead on the normal execution of the queue. The second algorithm, denoted `LinkedQ`, does persist the underlying node links. It reduces the number of fences by using a validity scheme to inform the recovery algorithm which nodes are adequate for recovery. It also adds a backward link to the queue nodes, for enabling to efficiently assist persisting concurrent operations.

We implemented these two algorithms on a platform with an Intel Cascade Lake processor and an Intel Optane NVRAM. Surprisingly, the new algorithms did not show a clear improvement over the state-of-the-art durable queue of Friedman et al. [16] although Friedman’s queue executes more blocking persist operations during the execution. Further investigations raised an interesting problem. Our queues frequently access flushed cache lines, and these accesses significantly deteriorated performance. It turned out that Intel flush instructions, which flush a cache line to the NVRAM, cause the flushed cache line to be invalidated in the cache, so that subsequent accesses yield cache misses and re-read the data from memory. (We tried various instructions including the most advanced `CLWB` instruction, but they all had the same performance degradation effect). The resulting additional loads from memory are significantly more costly on NVRAM than on DRAM, due to the high NVRAM read latency. While the recently-launched Intel Ice Lake processors with Optane persistent memory 200 series may provide flush instructions that do not invalidate the flushed cache lines, existing NVRAM architectures with Cascade Lake processors do not seem to support such instructions. Our impression is corroborated in the findings of [5, 15, 17, 20, 28, 50, 52]. Existing (costly) architectures will probably remain in use for years to come and one needs to use algorithmic modifications to obtain improved performance on such machines.

We amended the two algorithms further, obtaining algorithms that avoid accessing flushed locations. While changing the algorithms, we made sure that their original advantage of a single fence per update operation is maintained. An evaluation of this second amendment demonstrates a significant performance improvement, which confirms the high cost of accessing flushed content on these platforms.

The second contribution of this paper is a guideline for designing durable data structures and algorithms for NVRAM. In addition to the well-known guideline to minimize blocking persist operations, we recommend designing algorithms with reduced access to recently flushed cache lines¹. This guideline is relevant for platforms that invalidate cache lines when flushing their content to the memory, and the purpose is to avoid the cost of fetching data

¹We consider only explicitly flushed cache lines. There are additional implicit flushes, e.g., when the system evacuates cache lines to make space for new lines that need to be loaded to the cache. Such implicit flushes are hard to predict and this guideline does not attempt to consider them.

from the memory after it is evicted from the cache. This guideline is especially important in light of the high read latency of available NVRAM (see measurements in [50, 55]).

Our third contribution is the design of durable lock-free queues with significantly improved performance for the new Intel Optane architecture. We present `OptUnLinkedQ` and `OptLinkedQ`, obtained by amending `UnLinkedQ` and `LinkedQ` respectively according to the new guideline. `OptUnLinkedQ` and `OptLinkedQ` are the best performing lock-free durable queues available today. We compare the performance of `OptUnLinkedQ` and `OptLinkedQ` against state-of-the-art durable queues and against `UnLinkedQ` and `LinkedQ` themselves, which use minimal blocking persist operations but do not consider the new guideline and do not reduce access to flushed cache lines. While `OptUnLinkedQ` and `OptLinkedQ` outperform all other queues on nearly all thread counts and workloads, we believe `UnLinkedQ` and `LinkedQ` are still interesting to present. This is because for potential more advanced platforms that might provide flushing without cache invalidation, `UnLinkedQ` and `LinkedQ` may turn out best.

From a theoretical standpoint, it is interesting to note that `OptUnLinkedQ` and `OptLinkedQ` yield the best possible design characteristics for durability. Following our guideline above, they make zero accesses to content that was previously (explicitly) flushed, while they also meet the lower bound shown by Cohen et al. [9], executing only a single blocking persist operation per data structure update operation. Interestingly, while these theoretical characteristics are the best possible, they are also obtainable for any object. This follows from the universal construction of [9]. While Cohen’s universal construction of lock-free durably linearizable data structures is not practical, it has the above-mentioned characteristics (a single blocking persist instruction per update operation and no access to flushed content) and it is applicable to any object.

The rest of the paper is organized as follows. In Section 2 we elaborate on the model and the general upper bound on the design parameters. In Section 3 we recall the definitions of durable linearizability and lock-freedom as well as `MSQ`, the basic queue algorithm that we extend in our constructions. We discuss related work in Section 4. In Section 5 we provide an overview of the main ideas in the first amendment to `MSQ`: minimizing blocking persist operations, which produces `UnLinkedQ` and `LinkedQ`. In Section 6 we describe the second amendment to the two algorithms, adhering to the guideline of reducing access to flushed data, which results in the optimal queues `OptUnLinkedQ` and `OptLinkedQ`. The details of the `UnLinkedQ` algorithm are provided in Section 5, while further details of the rest of our queues are deferred to Appendices A–C. We argue about the durable linearizability and lock-freedom of our queues in Sections 7 and 8. The memory management scheme applied in our queues is described in Section 9, and the performance of all queue algorithms is evaluated in Section 10. We conclude in Section 11.

2 MODEL

In the persistent memory model, there are two levels of memory – volatile (registers, caches) and persistent (NVRAM). Values in the cache may be written back to the persistent memory implicitly by a cache eviction, or explicitly by flush instructions. We adopt the

failure model of Izraelevitz et al. [27] for crashes, which considers full-system crashes in which all processes fail together. The state of the volatile memory is lost in a crash, but the state of the persistent memory remains unaffected. After a crash, new threads are created and proceed with the computation. Each data structure may provide a recovery procedure to be invoked after the crash for restoring a consistent state of the object from its preserved state in the NVRAM. Our data structures apply a complete recovery before continuing with any new operation.

To maintain correctness in the presence of crashes, one has to ensure that necessary writes propagate from the cache to the persistent memory. To ensure a written value becomes persistent (after being written to the cache), one may issue a flush instruction and block until it completes. A flush instruction receives a memory address and flushes the content of the cache line containing this address to the persistent memory. Some flush instructions are asynchronous, enabling issuing multiple flushes concurrently as an optimization. Subsequently, a store fence instruction, denoted SFENCE (like the instruction name on Intel), may be placed to ensure completion of all previous asynchronous flushes. Throughout the paper, when mentioning a *persisting* of a location, we refer to an asynchronous flush of its address accompanied by an SFENCE to ensure that the data in this location has been written to the NVRAM.

Intel flush instructions (such as the synchronous CLFLUSH and the asynchronous CLFLUSHOPT and CLWB) take a memory location and write back the cache line containing it to the memory, if this line consists of modified data. According to the Intel architectures software developer’s manual [26], CLFLUSH and CLFLUSHOPT do not only write the cache line to the memory, but rather also invalidate it. Regarding CLWB, the Intel manual states that it *may* retain the line in the cache. However, on the Second Generation Intel Xeon Scalable Cascade Lake processor we use, CLWB seems to invalidate the cache line like CLFLUSHOPT does: replacing CLWB with CLFLUSHOPT in all the data structures we measured yielded similar performance. This is also noted by others [e.g. 5, 15, 17, 20, 28, 50, 52]. The recently-launched Third Generation Intel Xeon Scalable Ice Lake processors with Optane persistent memory 200 series may implement CLWB retaining lines in the cache, but NVRAM platforms currently in the market do not seem to support flushes without cache invalidation. Existing architectures will probably remain in use for years to come. Therefore, designers of efficient durable algorithms should take into consideration the cost of accessing a memory location after it was flushed and evicted from the cache.

To eliminate some of the costly persisting occurrences, we rely on the following assumption, which is based on the cache line granularity of write backs to memory. The assumption is mentioned in the SNIA NVM programming model [47, Section 10.1.1], adopted by Intel for working with persistent memory (as stated in Intel’s formal persistent memory programming book [44]), and is confirmed by Intel Senior Principal Engineer Andy Rudoff in online informal discussions [e.g. 4, 42, 43]. This assumption was also previously made in [6, Footnote 16] and [8, Assumption 2].

ASSUMPTION 1. *A cache line is evicted atomically to memory, thus, the order of multiple writes to the same cache line is preserved*

in memory. In other words, the content of a cache line in the memory reflects a prefix of the stores to that cache line.

As the order of writes to the same cache line is preserved in NVRAM², placing a flush plus SFENCE between them to ensure their persistence order (which is required for writes to different cache lines) is redundant.

In addition to a flush, another useful instruction for our algorithms is an instruction that writes back data directly to the memory without touching or polluting the cache (like `movnti`). Such asynchronous instructions require an accompanying SFENCE to ensure their completion.

2.1 Upper Bound on Accesses after a Flush

Due to cache invalidation after a flush, we recommend designing algorithms that minimize accesses to flushed content. This comes in addition to designing algorithms that minimize blocking flushes. In fact, we claim that it is possible to implement any object with a deterministic sequential specification in a durably linearizable lock-free way using the minimum possible number of fences (one per update operation and zero per read-only operation, as proved by [9]) while at the same time performing zero accesses to (explicitly) flushed cache lines.

To prove our claim, we leverage the universal construction of [9], called ONLL. ONLL consists of two main components. The first is a shared execution trace, containing a mark indicating the trace’s prefix guaranteed to be persistent. This prefix represents the current state of the object. The execution trace is not used during recovery, thus also not persisted to memory. The second component is local per-thread persistent logs (adopted from [8]), that will be read during recovery. An update operation first appends a record representing it to the execution trace, then appends a copy of the trace’s suffix that is not yet guaranteed to be persistent to its local log and persists it, and finally marks the trace’s prefix up to the current operation as persistent. A read-only operation calculates its response based on the current state of the object, represented by the trace’s marked prefix.

[9] proves that ONLL obtains the minimum possible number of fences. We suggest the following slight modification to ONLL: align log entries to cache lines, so that no two entries will share a cache line. By applying this modification, ONLL still obtains minimum fences, while also performing no access to flushed memory. This is because only data in the local per-thread persistent logs is explicitly flushed, and these logs’ cache lines are not accessed after their flush: they are read only during recovery, and not written by following log appends – which write to following cache lines thanks to our modification.

²Writes to the cache are not guaranteed to occur in program order, due to compiler optimizations, but program order can be enforced by placing inexpensive release fences (that prevent compiler optimizations, thus, ordering writes to cache). We placed release fences in our implementation where required, and we do not further mention them here.

3 PRELIMINARIES FOR THE DURABLE QUEUES

3.1 MS-Queue

Our persistent queue algorithms extend the widely used MSQ (the Michael and Scott queue [35]), a well-performing concurrent queue adequate for general hardware, included as part of the Java™ Concurrency Package [32]. This is a (non-persistent) lock-free FIFO queue, which supports enqueue and dequeue operations. It implements the queue as a singly-linked list with head and tail pointers. Nodes in the list have two fields: a value and a next pointer. The head points to the first node of the list, which functions as a dummy node. Subsequent nodes, after the dummy and until the node whose *next* pointer's value is NULL, contain the queue's items. The queue is initialized to an empty queue as a list that contains a single (dummy) node, to which both the head and tail point.

A dequeue operation checks if *next* of the obtained head is NULL (meaning the queue is empty). If so, this is a *failing dequeue* that returns without extracting an item from the queue. Otherwise, an attempt is made to update the head to point to its successive node in the list, using a CAS, and on failure the dequeue operation starts over. A dequeue that succeeds to perform a CAS that advances the queue's head is denoted a *successful dequeue*.

Enqueuing requires two CAS operations. Initially, a node with the item to enqueue is created. Then, an attempt to set *tail->next* to the address of the new node is made using a first CAS. The CAS fails if the value of *tail->next* is not NULL in that moment. In such a case, an attempt to advance *tail* to the current value of *tail->next* is made using a CAS, to help an obstructing enqueue operation complete. Then, a new attempt to perform the first CAS starts. After the first CAS succeeds, a second CAS is applied to update *tail* to point to the new node.

3.2 Linearizability and Durable Linearizability

Defining correctness for durable executions in the presence of both concurrency and NVRAM is not a trivial task. In this work, following recent work in this domain, we adopt durable linearizability [27] described below as a correctness criterion. Nevertheless, it is easy to verify that our proposed queues satisfy also other correctness criteria, like strict linearizability [1], persistent atomicity [18] and recoverable linearizability [3].

We recall some basic terminology. An *operation* consists of two events – *invocation* and *response*. An execution of a concurrent system in the full-system-crash model may be modeled by a finite sequence of events of three types: invocation events and response events, each tied to specific process and object, and system crash events (which are not tied to a specific process or object). Such sequence is denoted a *history*. An operation in a given history is *pending* if the history contains only its invocation event and not its response event. We refer to an operation for which the history contains also the response as *completed*. Each object has a *sequential specification*, which describes its behavior in sequential executions, where operations do not overlap.

A history without crash events is considered *linearizable* [22, 45] if each completed operation appears to take effect at once, between its invocation and its response events, in a way that satisfies the

sequential specification of the objects. Each pending operation is required to either take effect at once after its invocation in a way that satisfies the sequential specification of the objects, or not take effect at all. A history in the full-system-crash model (i.e., a history that might contain crashes in which all processes fail together and there is no subsequent thread reuse) is considered *durably linearizable* [27] if the history with the crashes omitted is linearizable.

3.3 Lock-Freedom

A concurrent object implementation is *lock-free* [21] if each time a thread executes an operation on the object, some thread (not necessarily the same one) completes an operation on the object within a finite number of steps. We extend the definition to executions with crashes, and define a concurrent object implementation to be lock-free in the presence of crashes if each time a thread executes an operation on the object, and there are no interrupting crash events since the operation's invocation, some thread (not necessarily the same one) completes an operation on the object within a finite number of steps. This definition is equivalent to the one brought in [57], which considers crashes as progress, as if a crash is one of the operations on the data structure. Lock-freedom guarantees system-wide progress. Our implementations are lock-free.

4 RELATED WORK

There has been a large body of work by multiple communities that provides algorithms for NVRAM. Several libraries for persistent transactional access to objects in NVRAM have been proposed [7, 10, 30, 33, 40, 49, 51, 53, 56], but persistent transactions require heavy-duty logging mechanisms, and thus do not yield highly efficient solutions, and are not competitive with ad-hoc constructions such as ours. [34] presents an NVRAM library taking another logging-based approach. Izraelevitz et al. [27] suggested to automatically make concurrent objects durably linearizable by adding a flush and a fence after each access to global memory (a read or a write). This transformation yields a durable variant of any existing lock-free data structure, but the resulting implementations are typically inefficient. The first ad-hoc efficient lock-free durable data structure was the queue presented by Friedman et al. [16], with a substantial reduction of the number of fences executed with each operation over the general construction of Izraelevitz. Subsequently, David et al. [12] presented lock-free durable set implementations (including a linked list, a skip list and a hash map). Zuriel et al. [57] improved over that construction and presented a set with a single SFENCE per update operation, thus meeting the lower bound of Cohen et al. [9] and also obtaining much better performance. Raad et al. [39] implemented a persistent FIFO queue to demonstrate the application of their suggested hardware model, but did not aim for optimized performance (e.g., they do not track the tail pointer, thus significantly slowing down enqueues).

5 FIRST AMENDMENT: QUEUES WITH MINIMUM FENCES

The current literature offers a fast queue with several fences [16] on the one hand, and a universal construction for all data structures with a single fence (per update operation) which is extremely

inefficient [9] on the other hand. A question that naturally arises is whether it is possible to reduce the number of fences to the minimum possible number, and at the same time be able to use this reduced blocking to obtain a better performing queue. In this section we provide two durably linearizable lock-free queues, `UnlinkedQ` and `LinkedQ`, that meet the theoretical lower bound on the number of fences (a single SFENCE per operation).

5.1 UnlinkedQ

As its name implies, `UnlinkedQ` does not rely on links between nodes for restoring the queue after a crash and therefore does not persist them, similarly to the basic idea in [57]. It keeps all information required for recovery in the nodes themselves, which are located in designated areas. Upon a crash, the recovery procedure checks these nodes to decide which ones are valid and belong to the resurrected queue. The links are still used to expedite operations on the queue when no crash occurs, but they are not required to reconstruct the queue after a crash. Care is taken to persist the queue order in the nodes to allow proper recovery.

`UnlinkedQ` places `index` and `linked` fields in each node to enable the recovery to identify which nodes in the designated areas should be restored and in what order. The `index` field states the node’s index in the queue (according to enqueue order). Overflow can be handled, but for now we allocate 64 bits for the `index` field and assume that it does not overflow (while humans are still around). The `linked` field marks nodes that have been added to the queue. After an enqueue succeeds to link a node to the queue, it sets its `linked` flag, and then persists the node content. The recovery procedure resurrects nodes that are marked `linked` and have an index larger than the head, and arranges them in the order induced by their indices. After advancing the head, a dequeuer persists the new head’s index, to indicate to the recovery that all nodes up to this one are dequeued. This scheme forms a consecutive prefix of dequeued nodes – all those that the head has persistently passed, thus satisfying the FIFO order requirement.

The simple scheme described so far involves several races which should be resolved. One race stems from the fact that the order in which enqueue operations complete does not necessarily match the linking order of their nodes. For example, it is possible that the enqueue of the fourth node in the queue has completed before the enqueue of the third node in the queue completed. Hence, it might be that the fourth node is marked `linked` while the third node is not. One consequence of this race is that the indices of valid linked nodes that the recovery identifies do not always form a sequence of consecutive integers. Even worse, a dequeue operation might point the head at a node inserted by a concurrent enqueue, whose content is not yet flushed and therefore contains a stale index that may confuse the recovery.

Next, we elaborate on the implementation of `UnlinkedQ`, including describing how it resolves the above-mentioned issues. The `UnlinkedQ` algorithm is presented in Figure 1. A description of its operations follows.

5.1.1 The Enqueue Operation. The enqueue operation first allocates a node and initializes its data (Lines 21–23). It then unsets `linked` (Line 24), sets the `index` of the new node to be the index of the last node plus one (Line 28), and attempts to link the node

Figure 1: `UnlinkedQ` implementation

```

1 class Node
2   Item* item
3   atomic<Node*> next
4   bool linked
5   int index

```

```

6 Item* Dequeue()
7   while (true)
8     head = Head
9     headNext = head.ptr->next
10    if (headNext == NULL)
11      FLUSH(&Head.index); SFENCE
12    return NULL
13    if (CAS(&Head, head, <headNext, headNext->index>))
14      dequeuedItem = headNext->item
15      FLUSH(&Head.index); SFENCE
16      if (nodeToRetire[tid]) // It equals NULL in the
17        first successful dequeue
18        retire(nodeToRetire[tid])
19      nodeToRetire[tid] = head.ptr
20    return dequeuedItem

```

```

20 Enqueue(item)
21   newNode = allocNode()
22   newNode->item = item
23   newNode->next = NULL
24   newNode->linked = false
25   while (true)
26     tail = Tail
27     if (tail->next == NULL)
28       newNode->index = tail->index + 1
29     if (CAS(&tail->next, NULL, newNode))
30       newNode->linked = true
31       FLUSH(newNode); SFENCE
32       CAS(&Tail, tail, newNode)
33     break
34   CAS(&Tail, tail, tail->next)

```

to the queue (Line 29). The reason `linked` is unset before `index` is updated, is that when the node is allocated, its `linked` flag might be set; thus, assigning the new node a relevant index in this state might erroneously cause the recovery to restore the node even though it is not yet linked to the queue.

After succeeding to link the node, the enqueue sets its `linked` flag (Line 30), to signal to the recovery (that would run if a crash occurs) that the node should be restored. The described order of writes to the node fields guarantees, based on Assumption 1³, that a node will be restored by the recovery only if it is successfully linked. Finally, the enqueue persists the node and advances the queue’s tail to point to the new node (Lines 31–32). If a concurrent enqueue operation interferes, the enqueue attempts to assist the other enqueue to advance the tail to point to its node (Line 34), before starting a new attempt to enqueue its own item.

³Applying Assumption 1 requires that the whole node resides on a single cache line, which is typically the case, and it also holds for the queues implemented in this paper. The method of [8] can be used to generalize the algorithms to nodes that span multiple cache lines without adding fence operations.

We note that the recovery procedure might restore a suffix of enqueues with nonconsecutive indices. This happens only if several enqueues are running when a crash occurs: an enqueue that linked e.g. the fourth node in the queue might have set its *linked* flag and persisted it before the crash, while an enqueue that linked the third node in the queue has not. Discarding pending enqueue operations which have not set and persisted the *linked* flag is correct due to the following observation:

OBSERVATION 1. *Durable linearizability allows pending operations to not be linearized. Therefore, the recovery may discard pending enqueue operations, which might result in a suffix of enqueued nodes with nonconsecutive indices.*

5.1.2 The Dequeue Operation. If a dequeue operation encounters an empty queue, it returns NULL. Otherwise, it attempts to advance the head by one node, and on success – it returns the oldest item to the caller. On failure it retries the whole scheme.

To signal to the recovery procedure that it should ignore the dequeued node, a successful dequeue operation ensures that the head’s index is persistently increased to a value bigger than or equal to its dequeued node’s index. Persisting the new head’s index is intended to indicate to the recovery not only that this node is dequeued, but also that all nodes up to this one are dequeued, and a failing dequeue also needs to persist the head’s index before returning in order to persist the previous dequeues that emptied the queue. This is obligatory due to the following observation:

OBSERVATION 2. *The recovery must restore a consecutive prefix of dequeued nodes, to satisfy the FIFO order requirement.*

The recovery achieves this by interpreting nodes with index smaller than or equal to the head’s index as dequeued.

A successful dequeue is responsible to reclaiming the node that was the head during the previous dequeue that this thread executed. This node to be retired is kept in a *nodeToRetire* array, consisting of a cell per thread. Its cells do not share cache lines to avoid false sharing. Each thread may access its cell using its thread ID as an index.

Next, we explain how dequeuers ensure that the correct head’s index is restored by the recovery. If we let a dequeuer persist the head’s address, and let the recovery determine the head’s index to be the *index* in the node pointed to by this head (as appears in NVRAM in the crash moment), then the recovery might erroneously restore a stale (smaller) head’s index value, and discard completed dequeues. This could happen if the enqueue of the node pointed to by the head has linked the node but was interrupted by the crash before persisting the node’s data. Therefore, `UnlinkedQ` takes a different approach to determine the head’s index in recovery.

`UnlinkedQ` makes the head hold not only a pointer to the dummy node, but also its index. They are held side-by-side and updated together atomically using a double-width CAS. A dequeuer starts by performing a double-width CAS (Line 13) that advances the head’s pointer and increments the head’s index. Next, the dequeuer persists the index placed in the head (Line 15). A failing dequeue assists persisting the head’s index too (Line 11).⁴ The recovery procedure

⁴We particularly specify the head’s index as the flushed value in order to stress that this is the data required in recovery, but its flush clearly writes the whole containing cache line to the memory.

restores the head’s index from the value kept in the queue’s head, rather than from the possibly stale value in the node pointed to by the head. This prevents discarding a completed dequeue: persisting the head’s index after incrementing it to the index of the dequeued node, makes the recovery procedure ignore the dequeued node.

The use of a double CAS can be eliminated (if the platform does not support it) by taking an alternative approach: Each thread could maintain a local index. After each time it advances the queue’s head, it would update the local index with the value of the new head’s index and persist it. The recovery would then restore the head’s index as the maximum across these local indices. The alternative handling of the head’s persistence described here, is actually required and applied in the second amendment of MSQ (see Section 6).

5.1.3 Recovery. The recovery procedure of `UnlinkedQ` resurrects nodes in the designated areas that are marked *linked* and have an *index* bigger than the head’s index. It then sets their links to form a linked list that holds the queue nodes in the order induced by their indices. This is implemented as follows.

The head’s index is not modified. A dummy node is allocated and assigned an index that matches the head’s index. The head’s pointer is set to point at this dummy node. Next, the recovery scans the designated areas and makes a list of recovered nodes, which are those with a set *linked* flag and an *index* larger than the head’s index. All other nodes are reclaimed. The recovered nodes are then sorted and their *next* pointers are set accordingly to create the queue. Finally, the queue’s tail is set to point to the last node in the queue.

We note that free nodes (owned by the memory manager) in the designated areas are appropriately ignored by the recovery: When the memory manager allocates a new designated area for nodes from the operating system, it zeros its content, to make all nodes consist of a zeroed index, and then persists it in NVRAM (by placing asynchronous flushes of the whole area accompanied by a single SFENCE). If the number of required nodes is unknown in advance, each time a designated area is depleted, the memory manager may allocate a new area from the operating system and initialize it in a similar manner using a single SFENCE. The zeroed indices guarantee that the unused nodes owned by the memory manager are ignored by the recovery. In addition to these not-yet-allocated nodes, nodes reclaimed by dequeuers are also ignored by the recovery thanks to their index value, as dequeue operations return nodes to the memory manager only after the head’s index persistently equals to the index of a subsequent node. Finally, nodes reclaimed by a previous recovery process are ignored thanks to either their *index* or their unset *linked*.

5.2 LinkedQ

`LinkedQ` also performs a single fence in each operation, but using a completely different approach. Here, we provide an overview of `LinkedQ`, and the full details appear in Appendix A.

The first idea `LinkedQ` employs is to make the recovery procedure able to deal with nodes whose data has not been persisted. This allows linking nodes to the queue without blocking to persist their data beforehand, thus avoiding one of the fences of the queue in [16]. To enable this, `LinkedQ` presents a mechanism that identifies nodes with stale data: a designated *initialized* flag in each node

signifies whether the content of the node is guaranteed to be valid. We maintain the invariant that if the node’s data is not initialized in NVRAM, then its *initialized* flag is unset in NVRAM. To achieve this, `LinkedQ`’s enqueue operation initializes the node in two steps: first, it initializes the node content, and then it sets the *initialized* flag. No SFENCE is issued during this execution, as Assumption 1 guarantees that the order of writes to the same cache line is not reversed.

For this scheme to work, we need to make sure that when a node is allocated, its *initialized* flag is unset. This can be easily done with an extra fence at allocation time, but would yield two fences per enqueue operation. We manage to avoid this fence by postponing the return of dequeued nodes to the memory manager. Think first of a simplified version that lets each thread accumulate k nodes it removed from the queue. After each k^{th} successful dequeue, before returning the k nodes to the memory manager, the thread clears their *initialized* flags, issues an (asynchronous) flush for each of the flags, and then a single blocking fence before letting the memory manager reclaim these objects. Such a simplified algorithm would execute $1 + 1/k$ fences per successful dequeue operation, not perfectly meeting the desired theoretical lower bound of a single fence. To reduce the number of fences to one, we take a more complex approach: After removing a node N from the queue, its dequeuing thread T clears N ’s *initialized* flag and records N ’s address for later. Instead of placing an additional fence every k dequeues, T will piggyback on the fence which its next successful dequeue anyhow performs: T will flush N ’s *initialized* flag before this fence, and return N to the memory manager after this fence. Such piggybacking on a fence of a later operation by the same thread makes sure that *initialized* flags are properly reset in memory before their nodes are reused, without incurring additional fences.

The recovery procedure resurrects all nodes reachable from the head through a path of consecutive nodes with the *initialized* flag set. It remains to ensure that completed enqueue operations are visible to the recovery procedure, even though previous nodes⁵ in the queue may have been enqueued by operations that have not yet completed. Before an enqueue operation completes, `LinkedQ` makes sure that all data on nodes from the head to the enqueued node is written back to the NVRAM. This guarantees that the recovery will reach the new node in its traversal from the head. Naively, before an enqueue operation completes, the enqueueer could traverse all nodes from the head until the new node, flush their contents, and then issue a single fence. This persists all relevant nodes but at a very high cost. To make this process efficient, we add a backward edge to the underlying linked list, and walk backwards persisting only nodes that might have not yet been persisted. We attempt to minimize the length of walk as much as possible. The full details of `LinkedQ` appear in Appendix A.

6 SECOND AMENDMENT: QUEUES WITH NO POST-FLUSH ACCESS

It turns out in the evaluation that reducing the number of fences is not enough to obtain high performance, and one should further improve the algorithms by reducing accesses to flushed data. In

⁵We think of the nodes as ordered by the underlying linked list of the queue. This order enables the terms *previous*, *preceding*, *subsequent*, *consecutive*, etc.

this section we describe further transformations of `UnlinkedQ` and `LinkedQ` into the new algorithms `OptUnlinkedQ` and `OptLinkedQ` respectively which do not access flushed locations, while still executing the minimal possible number of blocking fences per operation. Evaluation will show that the obtained algorithms yield excellent performance in current architectures. These algorithms are the fastest available persistent queues today, but we believe that `UnlinkedQ` and `LinkedQ` are of value on their own. This is because future architectures may provide flushes that do not invalidate cache lines. In such architectures `UnlinkedQ` and `LinkedQ` are expected to perform well thanks to using the minimal number of fence instructions. However, we cannot evaluate this performance prediction on the platform we currently possess.

6.1 OptUnlinkedQ

We provide an overview of `OptUnlinkedQ` here, and detail its pseudocode in Appendix B. We start with looking at what data is flushed in the `UnlinkedQ` algorithm, for use in a recovery. `UnlinkedQ` flushes the global head index, plus, the *index*, *item* and *linked* fields for each node in the underlying linked list. All of these values except for the *linked* field are later accessed. We eliminate these accesses using algorithmic modifications, amending `UnlinkedQ` to become `OptUnlinkedQ`.

First, we switch the global head index with a per-thread head index, holding the value that the head index had during the last dequeue by the thread. In `OptUnlinkedQ` the head pointer is a pointer only (with no adjacent index). Instead of persisting the global head index in the end of every dequeue operation as `UnlinkedQ` does, a dequeuer of `OptUnlinkedQ` copies the index value of the node pointed to by the head pointer to its local head index and persists it. In a recovery, the head index is set to the maximal index among the local head indices of all threads. Note that in this description we write to the local head index after persisting it. We eliminate this access in Section 6.3 below.

The *index* and *item* fields of a node in `UnlinkedQ` are written by the node’s enqueueer, and then (after the node is linked to the queue) – flushed by it, as well as read by subsequent operations: the *item* is read by a subsequent dequeue and the *index* is read by subsequent enqueueers. To prevent reads of a location after it is flushed, an enqueueer in `OptUnlinkedQ` physically splits the node into two nodes. The first one is called `Persistent` and it is flushed and not accessed after the flush. It is only used during a recovery, for which its content is essential. It is allocated in the designated areas that the recovery will scan. The second node is denoted `Volatile` and it is not flushed and not used in a recovery. However, `Volatile` is accessed after the flush of `Persistent` and is utilized to expedite the normal operation on the object. The *index* and *item* fields are placed in both `Persistent` and `Volatile`, with the two copies of each of them set to the same value. The enqueueer persists `Persistent`, while subsequent operations read the *index* and *item* from `Volatile`, thus adhering to our guideline. To enable access to the non-flushed fields, the queue’s head and tail point to the `Volatile` part.

Each part of the node contains additional fields other than *index* and *item*: The *linked* field is not accessed after the enqueueer performs the flush (except for during recovery), so there is no need

to keep two copies of it, and it is placed in `Persistent` only. The two following additional fields, which are not required in recovery, are placed in `Volatile`: `next`, and a pointer to the associated `Persistent` object, which the enqueuer sets for enabling the thread that reclaims the node later to locate `Persistent` and reclaim it together with `Volatile`.

The recovery procedure of `OptUnlinkedQ` resurrects `Persistent` objects in the designated areas that are marked *linked* and have an *index* bigger than the head's index. It then allocates matching `Volatile` objects and links them in a linked list in the order induced by their indices. This is implemented as follows.

Let *headIndex* be the maximal index among the local head indices of all threads. These per-thread indices are not modified. A dummy `Persistent` object is allocated and assigned the index *headIndex*. Next, the recovery scans the designated areas and makes a list of recovered `Persistent` objects, which are those with a set *linked* flag and an *index* larger than *headIndex*. All other `Persistent` objects are reclaimed. Then, in order to construct a queue of `Volatile` objects, for each of the recovered `Persistent` objects, as well as for the dummy `Persistent`, the recovery allocates a `Volatile` object and sets a pointer from it to the associated `Persistent` object. In addition, the *index* and *item* of each `Volatile` are copied from the associated `Persistent`. The `Volatile` objects are sorted by their indices, and their *next* pointers are set accordingly to create the queue. Finally, the queue's head and tail pointers are pointed at the first and last `Volatile` objects in the linked list.

6.2 OptLinkedQ

Transforming `LinkedQ` to a queue with no access to flushed data is trickier and involves further modifications, since it is problematic to eliminate accesses to a node's *next* field after its flush. It is easier to avoid accessing a node's backward link *pred* after its flush, so we make the recovery rely on the node's *pred* instead of *next*. Accordingly, the recovery mechanism is reversed, so that instead of resurrecting a path of consecutive valid nodes reachable from the head (as `LinkedQ` does), `OptLinkedQ` resurrects a chain of consecutive nodes reachable from the tail by backward links, ending with the node succeeding the dummy node. Similarly to `OptUnlinkedQ`, the queue node will be split into two nodes (`Persistent` and `Volatile`) so that the fields accessed after a flush (including the forward links) will not reside on the same cache line with the flushed fields (including the backward links).

Maintaining a single fence in each enqueue operation complicates the design of `OptLinkedQ` further: An enqueuer needs to use a single fence to ensure the persistence of both all recently inserted nodes and the tail. Therefore, before the final fence, the tail might be already persisted while some nodes are not, which may cause the recovery to encounter stale nodes when walking from the tail backwards. The way we deal with this problem is to let the recovery identify stale nodes during the traversal. When a stale node is discovered, the recovery starts over from an older recorded value of the tail, and repeats this process until finding a recorded tail value from which the node succeeding the head is reachable through a chain of persisted nodes. An *index* field placed in the nodes allows

the recovery to identify stale nodes. These are nodes whose *index* value is nonconsecutive. This field is set in a new node by its enqueuer, to the index of the last node plus 1.

The *index* field in nodes is also utilized to recover the head and the tail. As for the head, we cannot let dequeues flush the head pointer, because it will be accessed thereafter by following dequeues. Like in `OptUnlinkedQ`, we assign a per-thread head index, which dequeues update with the head index and persist, and recover the head index as the maximum among these values in all threads. The recovery terminates its backward walk when it reaches a node with the head index plus 1.

We can also not let enqueues flush the tail, because it will be accessed thereafter by subsequent enqueues. To solve this, we assign a per-thread last-enqueue pointer (pointing to the last `Persistent` object enqueued by the thread), as well as a per-thread last-enqueue index. Note that a backward walk from a last-enqueue pointer of a thread that performed an enqueue during the crash, might pass through stale nodes, as the per-thread last-enqueue pointer and index might be persisted before some queue nodes are persisted. Thus, the recovery looks for the per-thread last-enqueue pointer pointing to the latest node *up to which all nodes are persisted*. The recovery starts the traversal from the node pointed to by the per-thread last-enqueue pointer with the maximum associated per-thread last-enqueue index among all threads, and if the *index* of this node is different from the associated last-enqueue index, or if nonconsecutive *index* values are encountered (each of these cases implies that the inspected node is stale), it restarts the walk from the next last-enqueue pointer candidate, which is the one with the next largest associated index, until it identifies a `Persistent` object from which it establishes a complete walk up to the node succeeding the head.

The recovery scheme cannot be complete without dealing with the following rare scenario. All threads execute enqueues concurrently, the new last-enqueue pointer and index of them all are persisted in the memory, but then a crash occurs before any of the new nodes is persisted. In such a case, all last-enqueue pointers in all threads point to stale nodes, and the recovery will identify them as such. To restore a valid tail in this case, we assign *two* per-thread last-enqueue pointer and index, in which each thread keeps the details of both the last node enqueued by this thread and the penultimate node enqueued by this thread (up to which all queue nodes are definitely persisted by now because the penultimate enqueue was completed, including its fence instruction). The recovery sorts all last-enqueue indices (two of each thread) from largest to smallest and gathers their matching pointers to a single list of potential tail pointers. It attempts starting a backward walk from them, one after another. For each attempted tail pointer, if the index in the node it points to is different from the associated local enqueue index, or if a nonconsecutive index is encountered during the backward walk from it to the node with the recovered head index plus 1 (each of these cases implies that the index of the inspected node is stale) – the recovery moves on to try the next potential tail.

An enqueuer sets the *index* of the new node after setting its *item* and *pred*, so based on Assumption 1, when the recovery identifies the node's *index* as non-stale, it is guaranteed that its *item* and *pred* values are not stale. In this new recovery scheme that uses *index* to detect stale nodes, an *initialized* field like in `LinkedQ` is redundant.

Overall, the node's fields required in the recovery of `OptLinkedQ` are `index`, `item` and `pred`. A node of `OptLinkedQ` is composed of the following two parts: `Persistent` consists of the above mentioned fields, and `Volatile` consists copies of these fields for access after the flush of `Persistent`, as well as a `next` field that is not required in recovery, and a pointer to the associated `Persistent` object for its later reclamation. Additional details of `OptLinkedQ` appear in Appendix C.

6.3 Direct Write-Backs to Memory

The scheme described for `OptUnlinkedQ` replaces the global head index of `UnlinkedQ`, which is read, written and persisted for an unbounded number of times, with local variables that are never read (except for during recovery). However, they are still written and persisted for an unbounded number of times: each dequeue operation writes and persists the local head index of its thread. A standard write to a value that is absent from the cache causes a fetch of the containing cache line from the memory. Thus, we wish to avoid such a write to a flushed (thus evicted) location. Instead of a standard write, we issue a non-temporal write (using the `movnti` instruction) of the local head index, which writes back the value to the memory without touching the cache. This way, `OptUnlinkedQ` optimally performs no access to flushed cache lines.

To achieve this goal for `OptLinkedQ` as well, we need to eliminate any access to its local variables. The head index is handled just like in `OptUnlinkedQ`, using non-temporal writes. In addition, the local last-enqueue pointers and indices are also written and persisted for an unbounded number of times, and we update them too using non-temporal writes.

7 DURABLE LINEARIZABILITY

To define linearization points for our queue algorithms, we first define some supporting terminology. We start with *volatile linearization points*, which match the standard linearization points of `MSQ`, and are intuitively the steps applying the operations to the volatile queue. We also define a *survival point* for each operation, which marks the time from which the operation survives a crash. These two terms should basically be interpreted as: if an operation passes its survival point, then it is linearized at the time of its volatile linearization point. If it does not reach its survival point, then it is not linearized in this execution. Then, we derive the abstract state of the queue for each possible state of the queue's underlying list of nodes.

7.1 Linearization Points

DEFINITION 1 (VOLATILE LINEARIZATION POINT). *For each operation `op` in an execution `E` of the queue, we define its volatile linearization point to be the same as `op`'s standard linearization point in `MSQ`:*

- *Enqueue's volatile linearization point is the CAS that links its new node (Volatile object in case of `OptUnlinkedQ` and `OptLinkedQ`) to the last one.*
- *For a successful dequeue, its successful CAS that advances the queue's head is its volatile linearization point.*
- *The volatile linearization point of a failing dequeue is reading the next pointer of the dummy node (Volatile object in case*

of `OptUnlinkedQ` and `OptLinkedQ`), which is later revealed to be NULL.

An operation in `E` that does not reach its volatile linearization point as defined above, does not have a volatile linearization point (similarly to how not all operations in an execution have a linearization point). Intuitively, an operation's volatile linearization point is the step that applies the operation to the volatile queue.

DEFINITION 2 (SURVIVAL POINT). *For each operation `op` in an execution `E` of the queue, we define a survival point as follows:*

- **Successful Dequeue.** *Let `op` be a successful dequeue that advances the head to point to `N` at moment `t`. `op`'s survival point in `UnlinkedQ` and `LinkedQ` is the first (implicit or explicit) flush of the queue's head to the NVRAM after `t`, if a crash does not happen between `t` and this flush (else, the dequeue operation does not have a survival point). (Note that the flushed value of the head could be pointing to `N` or a subsequent node in the queue). `op`'s survival point in `OptUnlinkedQ` and `OptLinkedQ` is the first (implicit or explicit) flush of a per-thread head index to the NVRAM after `t` with a value greater than or equal to `N`'s index, if a crash does not happen between `t` and this flush (else, the dequeue operation does not have a survival point).*
- **Failing Dequeue.** *Let `op` be a failing dequeue. Let `head` be the last value read off the queue's head, before discovering the queue is empty. This read is followed by `op`'s volatile linearization point, where the next pointer in `head` is read and found NULL. Let `tℓ` be the time of this volatile linearization point, and let us look back in time at the point `t` where the value `head` was written to the queue's head. Let `tp` be the first time after `t`, where the content of the queue's head was flushed (implicitly or explicitly) to the memory, if a crash does not happen between `t` and this flush (else, `tp` is undefined, and so is the survival point of the dequeue). Then `op`'s survival point in `UnlinkedQ` and `LinkedQ` is defined to be the later between `tℓ` and `tp`. `op`'s survival point in `OptUnlinkedQ` and `OptLinkedQ` is defined similarly but using an alternative definition of `tp`, as the moment of the first (implicit or explicit) flush of a per-thread head index to the NVRAM after `t` with a value greater than or equal to `head->index`, if a crash does not happen between `t` and this flush (else, `tp` is undefined, and so is the survival point of the dequeue).*
- **Enqueue.** *Let `op` be an enqueue operation that inserts `N` to the queue. By `N` we refer to a Node object linked to the queue in case of `UnlinkedQ` and `LinkedQ`, and to a Persistent object pointed to by a Volatile object that is linked to the queue in case of `OptUnlinkedQ` and `OptLinkedQ`. Then the first of the following events to occur in `E` after the linking and before a crash occurs, is `op`'s survival point (if none of the following happens after the linking and before a crash, then the enqueue operation does not have a survival point):*
 - (1) *The queues differ in this event:*
 - *For `UnlinkedQ` and `OptUnlinkedQ`: An (implicit or explicit) flush of `N`'s linked field to the NVRAM after it is set to true.*
 - *For `LinkedQ`: The first time when all of the following conditions have been met, for some node preceding `N`,*

denoted dummy (intuitively, N has become reachable from dummy and marked as initialized in the NVRAM view):

- (a) The queue’s head has been flushed (implicitly or explicitly) to the NVRAM with a pointer to dummy. (Intuitively: dummy has become the queue’s dummy node in the NVRAM view.)
 - (b) The underlying linked list of the queue connects dummy to N ; and for each of the nodes along the way excluding N , its next field pointing to the subsequent node has been flushed (implicitly or explicitly) to the NVRAM. (Intuitively: N has been linked to the queue in the NVRAM view.)
 - (c) The setting of a true value to the initialized field in N reaches NVRAM by an (implicit or explicit) flush of N . (Intuitively: N has been marked as valid in the NVRAM.)
- For *OptLinkedQ*: The first time when all of the following conditions have been met, for some *Persistent* object preceding N , denoted dummy, and some *Persistent* object denoted last that is either N or a later *Persistent* object (intuitively, a backward path from the tail to the head through N became persistent):
- (a) Some per-thread head index has been flushed (implicitly or explicitly) to the NVRAM with the index of dummy (which means the head index will be recovered as dummy’s index or a bigger value).
 - (b) A last-enqueue pointer of some thread has been flushed (implicitly or explicitly) to the NVRAM with a pointer to last, and the associated last-enqueue index of that thread has been flushed to the NVRAM with the value last.index. (Intuitively: last has become a potential tail for the recovery.)
 - (c) The index of each *Persistent* object, from last backwards up to dummy excluding dummy, has been flushed (implicitly or explicitly) to the NVRAM with its final value (namely, the indices of all these *Persistent* objects have been flushed with consecutive values).
- (2) The survival point of a successful dequeue operation that dequeues the value inside N .

An operation in E that does not reach its defined-above survival point (in particular, a failing dequeue that does not reach both t_l and t_p , and an enqueue that does not reach any of the two detailed points), either due to a crash or since the execution ends, does not have a survival point.

Intuitively, an operation’s survival point is the flush that makes the operation survive a crash. The failing dequeue is somewhat different, as this operation does not modify the queue and we sometimes let its survival point be set to its volatile linearization point, rather than a flush. Operations that reach a survival point are linearized even if a crash occurs after their survival point before they complete. Note that for our queues the survival point always happens when the volatile linearization point has already occurred.

DEFINITION 3 (LINEARIZATION POINT). *The linearization point of an operation op in an execution E of the queue, is defined to be its volatile linearization point if op reaches a survival point in E . In this*

case, we say that op is linearized. Otherwise, op is not linearized, i.e., has no linearization point.

7.2 The Abstract State of the Queue

We define the abstract state of the queue at each moment (including during the recovery) in a given execution of each queue. This state reflects the applying of all operations linearized so far in their linearization order.

7.2.1 UnLinkedQ. The abstract head index in an execution of *UnLinkedQ* is set to the value⁶ of the *index* field in the queue’s head except in an interval before a crash. Between the last flush of the head to the NVRAM before a crash and the crash, the value of the abstract head index is not modified. It remains the value that was flushed to the memory.

The abstract state of the queue for execution E at moment t is defined as all items in nodes with indices bigger than the current abstract head index, which were enqueued by linearized enqueues whose linearization points occurred prior to t , ordered by their enqueues’ linearization order.

7.2.2 LinkedQ. The abstract head of the queue in an execution E of *LinkedQ* is defined similarly to the abstract head defined for *UnLinkedQ*, but this time we look at the head pointer. We define the abstract head to be the queue’s head value, except in an interval before a crash. From the last flush of the head (explicitly or implicitly) to the memory before a crash, until the crash, the abstract state of the head keeps the value flushed to the memory with no further abstract head state modifications in this interval.

Consider an execution E of the queue and a moment t during the execution, and consider the sequence of underlying list’s nodes, starting with the dummy node pointed to by the abstract head, and ending with the first node along the chain whose *next* pointer is NULL or points to a node enqueued by a non linearized enqueue. Namely, we do not include nodes whose enqueues have not been linearized yet. The abstract state of the queue for E at t is the sequence of items contained in all these nodes except for the first one (the dummy node). Note that the abstract state of the queue is an empty sequence if and only if the *next* pointer of the dummy node is NULL or points to a node enqueued by a non linearized enqueue.

7.2.3 OptUnLinkedQ. The abstract head index of the queue in an execution E of *OptUnLinkedQ* is set to the value of the *index* field in the node pointed to by the queue’s shared head, except in an interval enclosing a crash. Let *headIndex* be the biggest per-thread head index value flushed (explicitly or implicitly) to the NVRAM before the crash. Between the moment a pointer to a node with the index *headIndex* is written to the queue’s head and the moment the recovery procedure (that runs after the crash) terminates, the abstract head index keeps the value *headIndex*.

The abstract state of the queue for execution E at moment t is defined as all items in *Persistent* objects with indices bigger than the current abstract head index, which were enqueued by linearized

⁶To avoid confusion between the value in cache and the value in memory, we clarify that whenever a variable’s value is mentioned, we refer to the last value written to the variable (regardless of whether it has reached the NVRAM).

enqueues whose linearization points occurred prior to t , ordered by their enqueues' linearization order.

7.2.4 OptLinkedQ. The abstract head index of the queue in OptLinkedQ is defined exactly like that of OptUnlinkedQ. OptLinkedQ is our only algorithm for which the abstract state of the queue depends also on the abstract state of the tail. The abstract tail index in an execution E of OptLinkedQ is set to the index of the node enqueued by the last linearized enqueue operation. The abstract state of the queue is the sequence of items contained in the Persistent objects starting with the one enqueued by the last linearized enqueue and going through backward links until (including) the Persistent object with index bigger by 1 than the abstract head index, in reversed order; or an empty queue if the abstract tail index is not bigger than the abstract head index.

8 LOCK-FREEDOM

Our queue algorithms are lock-free: An operation might fail to perform a volatile linearization point only when another operation performs a conflicting volatile linearization point, causing the original operation to retry in a new loop iteration. We argue that at a crash-free interval of execution, it is guaranteed that within a finite number of retries, some operation succeeds to reach not only a volatile linearization point, but also a survival point, thus achieving a linearization point. Hence, system-wide progress is ensured.

The same basic argument applies to all operations of all presented queues: A queue operation op branches backwards and starts a new loop iteration each time another operation performs an obstructing volatile linearization point. If op does not succeed to pursue a volatile linearization point within n iterations, where n is the number of threads operating on the queue, then some other thread must have reached two volatile linearization points. This means it has completed the operation for which its first volatile linearization point was reached, and persisted it before returning (to satisfy durable linearizability). Thus, this operation is linearized. Yet, its linearization point might have occurred before op 's execution, and we need to verify that some linearization point occurs during op 's execution. We defer the details to Appendix D.

9 MEMORY MANAGEMENT

All queues evaluated in this paper (except for OneFileQ and RedoOptQ which were adopted from [40] and [11] respectively as they are with their integrated memory manager), use the same version of epoch based reclamation for memory management, called *ssmem*. This memory manager is adopted from [57], which implements a durable extension of the mechanism presented by [13] for volatile memory. *ssmem* maintains designated areas in the heap memory for node allocation. When a thread enqueues an item, it allocates a node from the next available space in these areas, or from a free list (to which dequeued nodes are inserted) if it is not empty. The memory manager keeps a persistent list of all the areas it allocated throughout the execution. During recovery, free lists are reconstructed from the unused chunks in these areas. Each thread in *ssmem* has its own allocator, operating on its separate designated areas and local free list, to avoid synchronization and reduce contention. See [57] for more details.

10 EVALUATION

Evaluated algorithms. We compare to the durable queue in [16] as the most efficient lock-free durably linearizable queue algorithm known today. However, the queue as presented in [16] is built to satisfy more than just durable linearizability. It contains a mechanism for retrieving previously obtained results after a crash, which is not required by durable linearizability, and is not provided by other durable data structures [12, 57]. To put all these data structures on the same level of guarantees, we remove the additional mechanism from [16], obtaining a thinner version of the original durable queue that executes faster, a version we denote DurableMSQ. Comparison to the exact original queue from [16] would yield better performance for us, but would not be fair. The extra mechanism in [16] can be easily added to the versions we propose (with the corresponding additional cost).

In addition, we compare to a persistent queue implementation resulting from applying the general construction of Izraelevitz [27] to MSQ. We also compare to the persistent queue version obtained by NVTraverse [15], which resembles IzraelevitzQ since the traversal phase in MSQ is empty, hence, the operations access directly the critical point, being the head or tail. The only difference between the two versions is that NVTraverseQ does not issue a fence after a flush that follows a read or CAS instruction. To complement the comparison, we compare to queues produced by wrapping a sequential queue implementation with a persistent transactional memory (PTM): OneFileQ, produced using the OneFile lock-free PTM [40], and RedoOptQ, produced using the RedoOpt PTM [11].

Platform. The queues were implemented in C++ and compiled using the g++ (GCC) compiler version 9.3.0 with a -O3 optimization level. We conducted our experiments on a machine running Linux (Ubuntu 18.04) equipped with 2 Intel Xeon Gold 6234 3.3GHz processors with 8 cores each. In experiments with up to 8 threads, each thread was attached to a different core of the same processor. In experiments with more than 8 threads in which the ninth and on threads were attached to the second processor, NUMA effects kick in impeding scalability and reducing performance, but the trends remain the same (OptUnlinkedQ performs best, OptLinkedQ is second best). To avoid NUMA effects, we utilize hyper-threading (SMT) on a single processor for measurements of more than 8 threads reported in Figure 2: we attach the $(8 + i)^{th}$ thread to the second virtual core on the same physical core as the i^{th} thread.

The machine has an L1 data cache of 32KB and an L2 cache of 1MB per core, and an L3 cache of 25MB per processor. It has 1.5TB of NVRAM (Intel Optane DC Persistent Memory), organized as 128GB DIMMs (6 per processor). The machine uses the NVRAM in App-Direct Mode Interleaved in our configuration. CLWB is utilized as a flush instruction, SFENCE as a store fence and movnti as a write-back to memory (non-temporal store) instruction.

Methodology. In each experiment, the queue is initialized with a certain number of enqueued items, and then operations are applied to it, for five seconds unless specified otherwise. Each data point $[x, y]$ in the graphs represents the average result of 10 experiments. In each experiment, x threads performed operations concurrently. The left graphs depict the throughput, namely, number of operations applied to each queue per second by the threads altogether.

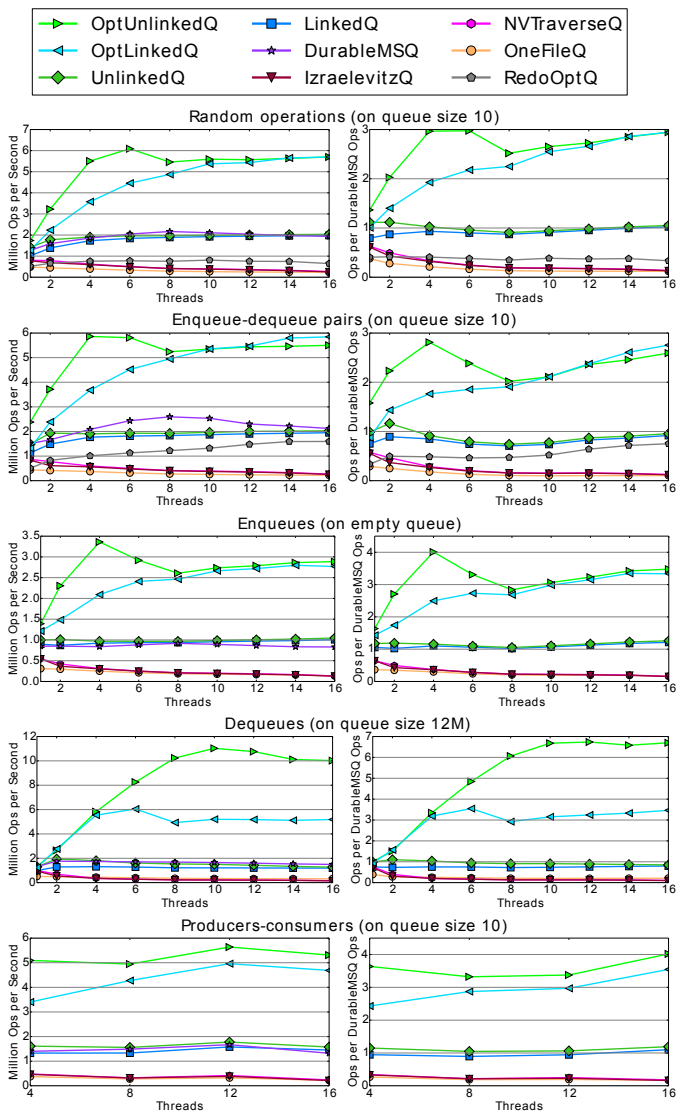


Figure 2: Measurement results

The right graphs depict the throughput ratio between each queue and the baseline DurableMSQ.

We ran various workloads following prior works (see Figure 2): In the first workload, operations were randomly chosen to be enqueue or dequeue (50-50 uniform distribution) following [23, 31, 54]. In the second workload, each thread ran enqueue-dequeue pairs, following measurements in [14, 16, 23, 31, 35, 36, 40, 54]. Next, we ran producers only (performing enqueues) on an empty queue. We also ran consumers only (performing dequeues) on a queue of size 12M following [38] for 1 second. At last, we ran a mixed producer-consumer workload, loosely following [19, 29, 38]. Here, unlike in other workloads, the threads did not run for a preset amount of time, but rather executed a preset number of operations: one quarter of the threads performed 1M dequeues and then 1M enqueues, and the rest performed 1M enqueues and then 1M dequeues. This is intended to ensure that the queue is not drained, as enqueues are

slower than dequeues. The initial queue in the presented graphs in the first, second and last workloads is of size 10. An initial size of 10K yields similar results (as we do not traverse the entire queue, but only touch the front and rear of the queue). RedoOpt is evaluated only in the first two workloads since we had problems running it on the other workloads.

Results. Our first two queue designs, UnlinkedQ and LinkedQ, perform better than DurableMSQ for some workloads and worse for others. They do not gain an advantage over DurableMSQ although performing minimum fences, due to accesses to flushed cache lines. Our efficient transformations that avoid such accesses, OptUnlinkedQ and OptLinkedQ, outperform all other queues including DurableMSQ, the state-of-the-art durable queue, in nearly all experiments. For example, OptUnlinkedQ runs more than twice faster than DurableMSQ for nearly all workloads with more than one thread. IzraelevitzQ is substantially slower than DurableMSQ and our queues, as expected from a universal construction that places many more fences than the tailor-made queues. NVTraverseQ, which is similar to IzraelevitzQ, shows nearly identical performance. The transactional approach of OneFileQ and RedoOptQ results in reduced performance as transactions impose additional overhead over a short operation.

11 CONCLUSION

In this paper we presented a new guideline for designing efficient durable algorithms suitable for the current architecture: reducing accesses to flushed memory. We demonstrated the advantage of following this guideline with durable queues. We first present novel queues that abide only to the known guideline of minimizing the fence count, meeting the theoretical lower bound on the number of fences from [9], executing only one blocking fence per operation. UnlinkedQ does not persist the links, but rather allocates the nodes on designated areas and adds an ordering mechanism, so the recovery procedure can look for valid nodes of the queue in the designated areas and order them correctly. LinkedQ uses a validity scheme on the queue nodes to inform the recovery algorithm which nodes are adequate for recovery, and adds a backward link to the queue’s underlying structure to allow enqueues to persist previous enqueues efficiently. These queues do not beat state-of-the-art queues in spite of issuing fewer fences. We then amended these queues to achieve zero accesses to flushed memory while still maintaining a single blocking fence per operation. The resulted queues demonstrate a significant performance improvement on the Intel Optane NVRAM over state-of-the-art durable queues, showing that, at least in our context, the second amendment is desirable.

REFERENCES

- [1] Marcos K Aguilera and Svend Frølund. 2003. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241* (2003). <https://hpl.hp.com/techreports/2003/HPL-2003-241.html>
- [2] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010). <https://doi.org/10.1109/JPROC.2010.2070830>
- [3] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2015. Robust shared objects for non-volatile main memory. In *OPODIS*. <https://doi.org/10.4230/LIPICs.OPODIS.2015.20>
- [4] Hadi Brais and Andy Rudoff. 2021. *Reply to On x86-64, is the "movnti" or "movntdq" instruction atomic when system crash?* <https://stackoverflow.com/a/65587308/7289606>

- [5] Heng Bu, Ming-Kai Dong, Ji-Fei Yi, Bin-Yu Zang, and Hai-Bo Chen. 2021. Revisiting persistent indexing structures on Intel Optane DC persistent memory. *Journal of Computer Science and Technology* 36, 1 (2021). <https://doi.org/10.1007/s11390-020-9871-0>
- [6] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices* 49, 10 (2014). <https://doi.org/10.1145/2714064.2660224>
- [7] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*. <https://doi.org/10.1145/1961295.1950380>
- [8] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols. *PACMPL* 1, OOPSLA (2017). <https://doi.org/10.1145/3133891>
- [9] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The inherent cost of remembering consistently. In *SPAA*. <https://doi.org/10.1145/3210377.3210400>
- [10] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *SPAA*. <https://doi.org/10.1145/3210377.3210392>
- [11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent memory and the rise of universal constructions. In *EuroSys*. <https://doi.org/10.1145/3342195.3387515>
- [12] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *USENIX ATC*. <https://usenix.org/conference/atc18/presentation/david>
- [13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ASPLOS*. <https://doi.org/10.1145/2786763.2694359>
- [14] Panagioti Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the combining synchronization technique. In *PPoPP*. <https://doi.org/10.1145/2370036.2145849>
- [15] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *PLDI*. <https://doi.org/10.1145/3385412.3386031>
- [16] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *PPoPP*. <https://doi.org/10.1145/3178487.3178490>
- [17] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making lock-free data structures persistent. In *PLDI*. <https://doi.org/10.1145/3453483.3454105>
- [18] Rachid Guerraoui and Ron R Levy. 2004. Robust emulations of shared memory in a crash-recovery model. In *ICDCS*. <https://doi.org/10.1109/ICDCS.2004.1281605>
- [19] Andreas Haas, Michael Lippautz, Thomas A Henzinger, Hannes Payer, Ana Sokolova, Christoph M Kirsch, and Ali Sezgin. 2013. Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation. In *CF*. <https://doi.org/10.1145/2482767.2482789>
- [20] Xiangpeng Hao. 2019. *Is CLWB actually implemented?* <https://blog.haoxp.xyz/posts/is-clwb-implemented>
- [21] Maurice Herlihy. 1991. Wait-free synchronization. *TOPLAS* 13, 1 (1991). <https://doi.org/10.1145/114005.102808>
- [22] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990). <https://doi.org/10.1145/78969.78972>
- [23] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The baskets queue. In *OPODIS*. https://doi.org/10.1007/978-3-540-77096-1_29
- [24] IBM. [n.d.]. *IBM MQ*. <https://ibm.com/software/products/en/ibm-mq>
- [25] Intel. 2019. *3D XPoint™: A breakthrough in non-volatile memory technology*. <https://intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html>
- [26] Intel. 2020. *Intel® 64 and IA-32 architectures software developer's manual*. <https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf>
- [27] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*. https://doi.org/10.1007/978-3-662-53426-7_23
- [28] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *SoCC*. <https://doi.org/10.1145/3419111.3421294>
- [29] Christoph M Kirsch, Michael Lippautz, and Hannes Payer. 2013. Fast and scalable, lock-free k-FIFO queues. In *PACT*. https://doi.org/10.1007/978-3-642-39958-9_18
- [30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. In *ASPLOS*. <https://doi.org/10.1145/2872362.2872381>
- [31] Edya Ladan-Mozes and Nir Shavit. 2004. An optimistic approach to lock-free FIFO queues. In *Distributed Computing*. <https://doi.org/10.1007/s00446-007-0050-0>
- [32] Doug Lea. 2009. The Java concurrency package (JSR-166).
- [33] Virendra Marathe, Achin Mishra, Ameer Trivedi, Yihe Huang, Faisal Zaghoul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, et al. 2018. Persistent memory transactions. *arXiv preprint* (2018). arXiv:1804.00701
- [34] Amiraman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and fast persistence for volatile data structures. In *ASPLOS*. <https://doi.org/10.1145/3373376.3378456>
- [35] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*. <https://doi.org/10.1145/248052.248106>
- [36] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *PPoPP*. <https://doi.org/10.1145/2442516.2442527>
- [37] Oracle. [n.d.]. *Oracle Tuxedo Message Queue*. https://docs.oracle.com/ed/E35855_01/otmq/docs12c/overview/overview.html
- [38] Or Ostrovsky and Adam Morrison. 2020. Scaling concurrent queues by using HTM to profit from failed atomic operations. In *PPoPP*. <https://doi.org/10.1145/3332466.3374511>
- [39] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *PACMPL* 4, POPL (2020). <https://doi.org/10.1145/3371079>
- [40] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A wait-free persistent transactional memory. In *DSN*. <https://doi.org/10.1109/DSN.2019.00028>
- [41] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008). <https://doi.org/10.1147/rd.524.0465>
- [42] Andy Rudoff. 2019. *Reply to How to use CLWB instructions*. <https://groups.google.com/g/pmem/c/R8H3sKq9sLQ/m/tL7Kng4BAAJ>
- [43] Andy Rudoff. 2020. *Reply to 8 byte atomicity & larger store operations*. https://groups.google.com/g/pmem/c/6_5daOuEI00/m/nY_mtKd0CAAJ
- [44] Steve Scargall. 2020. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature. <https://doi.org/10.1007/978-1-4842-4932-1>
- [45] Gal Sela, Maurice Herlihy, and Erez Petrank. 2021. Brief announcement: Linearizability: A typo. In *PODC*. <https://doi.org/10.1145/3465084.3467944>
- [46] Gal Sela and Erez Petrank. 2021. Durable queues: The second amendment. In *SPAA*. <https://doi.org/10.1145/3409964.3461791>
- [47] SNIA. 2017. *NVM Programming Model (NPM)*. https://snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf
- [48] Pivotal Software. [n.d.]. *RabbitMQ*. <https://rabbitmq.com>
- [49] Intel PMDK team. [n.d.]. *Persistent memory programming*. <https://pmem.io>
- [50] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *DaMoN*. <https://doi.org/10.1145/3329785.3329930>
- [51] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ASPLOS*. <https://doi.org/10.1145/1950365.1950379>
- [52] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L Scott. 2020. Montage: A general system for buffered durably linearizable data structures. *arXiv preprint* (2020). arXiv:2009.13701
- [53] Kai Wu, Jie Ren, and Dong Li. 2019. Architecture-aware, high performance transaction for persistent memory. *arXiv preprint* (2019). arXiv:1903.06226
- [54] Chaoran Yang and John Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *PPoPP*. <https://doi.org/10.1145/2851141.2851168>
- [55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *FAST*.
- [56] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. 2019. Optimizing persistent memory transactions. In *PACT*. <https://doi.org/10.1109/PACT.2019.00025>
- [57] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *PACMPL* 3, OOPSLA (2019). <https://doi.org/10.1145/3360554>

Figure 3: LinkedQ implementation

```

35 class Node
36     Item* item
37     atomic<Node*> next
38     atomic<Node*> pred
39     bool initialized

```

```

40 Item* Dequeue()
41     while (true)
42         head = Head
43         headNext = head->next
44         if (headNext == NULL)
45             FLUSH(&Head); SFENCE
46             return NULL
47         if (CAS(&Head, head, headNext))
48             dequeuedItem = headNext->item
49             if (nodeToPersistAndRetire[tid])
50                 FLUSH(&nodeToPersistAndRetire[tid]->
                    initialized)
51             FLUSH(&Head)
52             SFENCE
53             headNext->pred = NULL
54             if (nodeToPersistAndRetire[tid])
55                 retire(nodeToPersistAndRetire[tid])
56             head->initialized = false
57             nodeToPersistAndRetire[tid] = head
58             return dequeuedItem

```

```

59 FlushNotPersistedSuffix(notPersisted)
60     do
61         FLUSH(notPersisted)
62         notPersisted = notPersisted->pred
63     while (notPersisted != NULL);
64 Enqueue(item)
65     newNode = allocNode()
66     newNode->item = item
67     newNode->next = NULL
68     newNode->initialized = true
69     while (true)
70         tail = Tail
71         if (tail->next == NULL)
72             newNode->pred = tail
73             if (CAS(&tail->next, NULL, newNode))
74                 FlushNotPersistedSuffix(newNode)
75                 SFENCE
76                 CAS(&Tail, tail, newNode)
77                 // All nodes preceding newNode are persistent
78                 newNode->pred = NULL
79                 break
80     CAS(&Tail, tail, tail->next)

```

A LINKEDQ DETAILS

The LinkedQ algorithm appears in Figure 3. Next, we describe its operations in detail.

A.1 The Enqueue Operation

The enqueue operation first allocates a node denoted *newNode* from the memory manager and initializes its data (Lines 65–67). Then it sets *newNode*'s *initialized* flag (Line 68). In what follows,

the enqueue operation attempts to link *newNode* to the last node (Line 73). Note that it might have just linked a node whose data is not persisted in NVRAM. If the link to *newNode* is written back to NVRAM (which could happen implicitly due to a cache eviction) and then a crash occurs, the recovery would have to deal with reaching a node with stale data. The correctness is maintained using the *initialized* flag and a matching recovery procedure: The *initialized* flag is used as a stamp indicating to the recovery that *newNode* is initialized. Relying on Assumption 1, the order of writing *initialized* after *newNode*'s data is preserved in NVRAM. Accordingly, the recovery resurrects only nodes with the *initialized* flag set. This guarantees that it resurrects only nodes with persisted data. If a crash occurs after the link to *newNode* is flushed to NVRAM and before *newNode*'s data is written back to NVRAM, then *newNode*'s *initialized* flag must be unset in NVRAM, thus, the recovery will ignore it (and all nodes linked after it).

The recovery procedure resurrects all nodes reachable from the head through a path of consecutive nodes with the *initialized* flag set. Since durable linearizability allows the recovery procedure to ignore enqueue operations that are concurrent with a crash and elide their nodes from the queue, the fact that the recovery procedure ignores nodes of ongoing enqueues that were linked after a node with an unset *initialized*, does not break durable linearizability. However, after an enqueue operation completes, the recovery procedure must not discard it, even if earlier nodes belong to incomplete enqueue operations. To this end, after successfully linking *newNode*, the enqueue operation ensures that the path of nodes leading from the head to *newNode* is persisted (Lines 74–75). This could be achieved naively by flushing all nodes from the head until *newNode*. To save redundant flushes, an enqueue operation avoids flushing a prefix of queue's nodes that are guaranteed to be already flushed. Instead, it flushes only a suffix of queue's nodes that are not guaranteed to be persistent. To identify the relevant suffix, we place backward links in the nodes, but we remove a backward link when we know that all previous nodes in the queue have already been persisted. The backward links preserve the following invariant: all queue nodes (starting from the current queue's head) that precede a node with a nullified backward link have all relevant content (their item, set *initialized* flag and a non-NULL forward link) persisted.

To maintain a backward path connecting the linked list's nodes that should be flushed, an enqueue operation links a node with a backward link pointing to the previous node (Line 72). After linking *newNode*, its enqueue operation traverses the queue from *newNode* backwards using the backward links, until reaching a NULL backward link, and flushes the content of all traversed nodes (including *newNode* itself) (Lines 60–63). Finally, it issues a single SFENCE to block until all these flushes complete (Line 75). By the above-mentioned invariant, all nodes starting from the current head and preceding this suffix of nodes, are persistent. Now, as this suffix is persisted as well, the data of all nodes preceding *newNode* starting from the current head is guaranteed to be persistent. As an optimization to prevent future enqueues from flushing these nodes, the enqueue operation then sets *newNode*'s backward link to NULL (Line 78). Thus, each enqueue operation that reaches *newNode* from now on, during its backward walk, would not need to traverse the preceding persistent nodes. Note that backward links are not used in the recovery and there is no need to explicitly flush them.

To complete the enqueue operation, the tail is advanced to point to *newNode* (Line 76). Like in the original MSQ, a concurrent enqueue might prevent the enqueue’s linking. In this case, the enqueuee tries to assist and advance the tail to point to the node enqueuee by the obstructing enqueue (Line 80), before starting a new attempt to enqueue its own item.

We note that, as an optimization on x86 platforms, the SFENCE in Line 75 can be eliminated, because the following CAS instruction serves as an SFENCE guaranteeing completion of previous flushes. In the measured implementations of all algorithms, each SFENCE preceding a CAS is eliminated. We did not include this optimization in the paper’s pseudocode for clarity.

A.2 The Dequeue Operation

The dequeue operation attempts to extract the oldest item, placed in the node subsequent to the dummy node. If the queue is empty when the dequeue operation takes effect, it returns NULL. But before returning, the failing dequeue must persist the head (Line 45), to ensure that previous ongoing dequeues that emptied the queue are persistent. Otherwise, if a crash occurs after the failing dequeue returns, the previous dequeues might be discarded. This would break durable linearizability, since it will be impossible to linearize the completed failing dequeue correctly as applied to an empty queue, without the previous dequeues being linearized beforehand.

If the queue is not empty, the dequeuer attempts to advance the head by one node (Line 47), and on success – returns the oldest item to the caller. On failure it retries the whole scheme. Before returning, the dequeuer persists the head (Lines 51–52), to comply with durable linearizability, which requires that completed operations be linearized.

Each dequeue makes sure that the dummy node from which it advances the head will be unreachable by future operations, so that the next successful dequeue by the same thread will safely return this dummy node to the memory manager. To make it unreachable by backward walks (of enqueue operations that will try to identify a not persisted suffix), the dequeuer disconnects the backward link from the new dummy head to the previous one (Line 53). In addition, persisting the head guarantees that the previous dummy node will be unreachable by future operations even in case of a crash.

A successful dequeue does not simply return the previous dummy node (i.e., the node from which the previous successful dequeue by the same thread has advanced the head) to the memory manager as it is. Recall from Section 5.2 that we must make sure that newly allocated nodes have their *initialized* flags reset. The *initialized* flag placed in each node is used by its enqueuee to signal to the recovery when the node’s data is persisted. Suppose a node is erroneously allocated in an enqueue operation with a set *initialized* flag. After the enqueue operation links the node, the link to the node might be implicitly flushed to the NVRAM, and – before the node’s data is persisted – a crash might follow. The recovery procedure would then find the linked node, containing stale data including a set *initialized* flag, and would erroneously interpret the node with the stale content as part of the queue. To prevent this scenario, enqueuees could unset the *initialized* flag after the node’s allocation and then persist it before initializing its data, but this incurs an additional fence. Instead, we make sure that a node is

always allocated with an *initialized* flag persistently unset. Next we explain how we ensure that.

If we allocate nodes from the operating system, we would get nodes with arbitrary content, possibly with the *initialized* field set. Instead, we implement a memory manager that maintains large designated areas from which all node allocations are performed.

First, we explain how nodes, allocated from the designated areas for the first time, are allocated with a persistently unset *initialized* value. If the number of nodes required by the program is known in advance, then on program startup, the memory manager may allocate a sufficiently large designated area for nodes from the operating system, zero its content to make all nodes marked as not initialized, and then persist it in NVRAM (by placing asynchronous flushes of the whole area accompanied by a single SFENCE). This guarantees that when the memory manager allocates a node for the first time, its *initialized* field is unset. If the number of required nodes is unknown in advance, each time a designated area is depleted, the memory manager may allocate a new area from the operating system, and initialize it in a similar manner using a single SFENCE.

It remains to explain how nodes, reallocated from the designated areas after reclamation, are allocated with an *initialized* flag persistently unset. The dequeue operation and the recovery procedure return nodes to the memory manager, hence, they are responsible to return them with an *initialized* flag persistently unset.

Starting with dequeue, a successful dequeuer could unset the *initialized* flag of the dummy node from which it has advanced the head and then perform additional flush and SFENCE to persist the unset *initialized* flag before returning the node to the memory manager. However, to achieve the fence lower bound of a single SFENCE per operation, LinkedQ takes a different approach.

The persistence of the previous dummy node’s *initialized* flag is accomplished through piggybacking on the next successful dequeue’s SFENCE, which this thread is anyhow going to execute (in Line 52). More precisely, the dequeuer sets the previous dummy node’s *initialized* flag to *false* (Line 56) after the queue’s head persistently points to a subsequent node. The dequeuer thread postpones the reclamation of this previous dummy node, and keeps the node locally in a *nodeToPersistAndRetire* array (Line 57). This array consists of a pointer cell per thread, each cell lying in another cache line to avoid false sharing. Each thread may access its cell using its thread ID as an index. In the next successful dequeue execution of the same thread, right before its SFENCE, the *initialized* flag of the node we kept aside is flushed (Line 50). After the fence completes, the node may be returned to the memory manager (Line 55).

As for the recovery, as detailed in Appendix A.3, for each node with a set *initialized* flag that it returns to the memory manager – the recovery unsets the flag and flushes it. A single SFENCE placed in the end of the recovery ensures that these flags are unset in the memory.

A.3 Recovery

The recovery procedure of LinkedQ, running after a crash, resurrects all nodes reachable from the head through a path of consecutive nodes with the *initialized* flag set. It does so by leaving the queue’s head as it is and reconstructing the queue as follows.

- (1) If the *initialized* flag of the dummy node (namely, the node pointed to by the head) is unset, the recovery procedure sets

Figure 4: OptUnlinkedQ implementation

```

81 class Persistent
82     Item* item
83     int index
84     bool linked
85 class Volatile
86     Item* item
87     int index
88     atomic<Volatile*> next
89     Persistent* persistentNode

```

```

90 Item* Dequeue()
91     while (true)
92         head = Head
93         headNext = head->next
94         if (headNext == NULL)
95             movnti(&localData[tid].headIndex, head->index)
96             SFENCE
97             return NULL
98         if (CAS(&Head, head, headNext))
99             dequeuedItem = headNext->item
100            movnti(&localData[tid].headIndex, headNext->
101                index)
102            SFENCE
103            if (localData[tid].nodeToRetire)
104                retire(localData[tid].nodeToRetire->
105                    persistentNode)
106                retire(localData[tid].nodeToRetire)
107            localData[tid].nodeToRetire = head
108            return dequeuedItem

```

```

107 Enqueue(item)
108     newNode = allocVolatile()
109     newNode->item = item
110     newNode->next = NULL
111     newNode->persistent = allocPersistent()
112     newNode->persistent->item = item
113     newNode->persistent->linked = false
114     while (true)
115         tail = Tail
116         if (tail->next == NULL)
117             newNode->persistentNode->index = tail->index + 1
118             newNode->index = newNode->persistentNode->index
119             if (CAS(&tail->next, NULL, newNode))
120                 newNode->persistentNode->linked = true
121                 FLUSH(newNode->persistentNode); SFENCE
122                 CAS(&Tail, tail, newNode)
123                 break
124     CAS(&Tail, tail, tail->next)

```

the dummy node's *next* to NULL and then sets its *initialized* flag. The order of the last two writes ensures (based on Assumption 1) a proper recovery from a possible crash in the midst of the current recovery. The tail is set to point to the dummy node as well.

- (2) Otherwise (the dummy node's *initialized* is set) –
- (a) The recovery procedure traverses the nodes starting with the one pointed to by the head, until it reaches either a node whose *next* value is NULL, or a node with an unset

initialized. In the first case, the recovery points the queue's tail to the last traversed node.

- (b) If the traversal ends due to a node with an unset *initialized* flag, then let *P* be its preceding node. The recovery sets *P.next* to NULL and flushes it, and sets the tail to point to *P*.

In all cases, the *pred* field of the last node (pointed to by the tail) is set to NULL. In addition, throughout the queue traversal, the addresses of all traversed nodes with a set *initialized* flag are recorded. All other nodes in the designated allocation areas are reclaimed. For each reclaimed node with a set *initialized* flag, the recovery unsets the *initialized* flag and flushes it before retiring the node. There could be at most two such nodes per thread: There is at most one such node (namely, a node which is not part of the queue but has a set *initialized* flag) which the thread has dequeued and placed in its local *nodeToPersistAndRetire* array, where the node awaits its persistence. In addition, there could be another such node – a node that the thread was about to enqueue, if the thread were in the middle of an enqueue operation when the crash occurred; or alternately a node that the thread has just advanced the head from, if the thread were in the middle of a dequeue operation when the crash occurred.

If any flush were executed during the recovery, a single SFENCE is placed in the end to ensure the completion of the executed flushes.

B OPTUNLINKEDQ DETAILS

Figure 4 contains the pseudocode of the OptUnlinkedQ algorithm, described in Section 6.1. Note that the queue's global head and tail pointers point to Volatile nodes. The *movnti* instruction is a non-temporal store instruction that writes back data directly to the memory, bypassing the caches.

C OPTLINKEDQ DETAILS

The pseudocode of the OptLinkedQ algorithm appears in Figures 5 and 6. The queue's global head and tail pointers point to Volatile nodes. *localData* is an array consisting of a cell per thread. Each cell consists of the fields *headIndex* and *nodeToRetire* accessed in dequeues, and *lastEnqueues* (an array containing two cells, each composed of a pointer to a Persistent object and an index), *lastEnqueuesIndex* and *validBit* accessed in enqueues. *localData* array's cells do not share cache lines to avoid false sharing. In addition, for each cell, the *lastEnqueues* array and *headIndex* field, which are written using *movnti* instructions, are kept in a cache line separate from the rest of the cell's fields.

Next, we describe OptLinkedQ's operations in detail.

C.1 The Enqueue Operation

The enqueue operation first allocates a Volatile node denoted *newNode* from the memory manager and a matching Persistent node and initializes their data (Lines 171–175). Then, before attempting to link *newNode* to the last node, it sets the *pred* and *index* fields of both the Volatile and Persistent parts (Lines 179–182). The *index* field of the Persistent object serves as a stamp indicating to the recovery that the object's data is up-to-date: *index* is the last written field of the Persistent object, for ensuring that if this object is traversed during a recovery walk, and its *index* is

Figure 5: OptLinkedQ implementation – Objects and Dequeue

```

125 class Persistent
126     Item* item
127     Persistent* pred
128     int index
129 class Volatile
130     Item* item
131     atomic<Volatile*> next
132     atomic<Volatile*> pred
133     int index
134     Persistent* persistentNode

```

```

135 Item* Dequeue()
136     while (true)
137         head = Head
138         headNext = head->next
139         if (headNext == NULL)
140             movnti(&localData[tid].headIndex, head->index)
141             SFENCE
142             return NULL
143         if (CAS(&Head, head, headNext))
144             dequeuedItem = headNext->item
145             movnti(&localData[tid].headIndex, headNext->
146                 index)
146             SFENCE
147             headNext->pred = NULL
148             if (localData[tid].nodeToRetire)
149                 retire(localData[tid].nodeToRetire->
150                     persistentNode)
150                 retire(localData[tid].nodeToRetire)
151             localData[tid].nodeToRetire = head
152             return dequeuedItem

```

identified as non-stale, then all the object’s data is non-stale. This is due to Assumption 1, guaranteeing that the order of writing *index* after the other fields is preserved in NVRAM.

Next, the enqueue operation attempts to link *newNode* to the last *Volatile* node (Line 183), and on success it advances the queue’s tail and ensures that the path of nodes leading from the head to *newNode->persistentNode* is flushed to the NVRAM (Lines 184–185).

It then records the address and index of the newly enqueued Persistent node in the thread’s *lastEnqueues* array (Line 186). This array contains two cells per thread – for keeping record of the thread’s last and penultimate enqueued nodes. The thread writes alternately – on each enqueue it writes to the cell with index *localData[tid].lastEnqueueIndex* and in the end flips its *lastEnqueueIndex* (in Line 169). The writes to *lastEnqueues* are performed using *movnti* instructions (Lines 166–167). In case a crash occurs after only one of the address and index was written to the memory, the subsequent recovery needs to identify that the cell’s content is invalid and should be ignored. To this end, we place a valid bit in both the address and value (the least significant bit of the address and the most significant bit of the index). A *lastEnqueues* cell is considered valid only if the valid bits of its address and index match. After the writes, the value of *localData[tid].validBit* is flipped if *localData[tid].lastEnqueues=1* (Line 168), so that the thread’s following writes to its two *lastEnqueues* cells will be with the opposite valid bit value.

Figure 6: OptLinkedQ implementation – Enqueue

```

153 FlushNotPersistedSuffix(notPersisted)
154     while (true)
155         pred = notPersisted->pred
156         if (pred == NULL)
157             break
158         FLUSH(notPersisted->persistentNode)
159         notPersisted = pred
160 ZeroBit(value, bitIndex)
161     return value & ~(1 << bitIndex)
162 ApplyBit(value, bitIndex, bitValue)
163     return ZeroBit(value, bitIndex) | (bitValue <<
164         bitIndex)
164 RecordLastEnqueue(newNode)
165     i = localData[tid].lastEnqueuesIndex
166     movnti(&localData[tid].lastEnqueues[i].ptr, ApplyBit(
167         newNode->persistentNode, 0, localData[tid].
168         validBit))
167     movnti(&localData[tid].lastEnqueues[i].index, ApplyBit(
168         newNode->index, sizeof(newNode->index)*8-1,
169         localData[tid].validBit))
168     localData[tid].validBit ^= i // Flip valid bit if i=1
169     localData[tid].lastEnqueuesIndex ^= 1 // Flip index
170 Enqueue(item)
171     newNode = allocVolatile()
172     newNode->item = item
173     newNode->next = NULL
174     newNode->persistentNode = allocPersistent()
175     newNode->persistentNode->item = item
176     while (true)
177         tail = Tail
178         if (tail->next == NULL)
179             newNode->pred = tail
180             newNode->index = tail->index + 1
181             newNode->persistentNode->pred = tail->
182                 persistentNode
182             newNode->persistentNode->index = newNode->index
183             if (CAS(&tail->next, NULL, newNode))
184                 CAS(&Tail, tail, newNode)
185                 FlushNotPersistedSuffix(newNode)
186                 RecordLastEnqueue(newNode)
187                 SFENCE
188                 // All nodes up to newNode are persistent
189                 newNode->pred = NULL
190                 break
191             CAS(&Tail, tail, tail->next)

```

Finally, the enqueue operation issues an SFENCE (Line 187) to ensure the completion of all executed flushes and *movnti* instructions. In particular, all Persistent nodes succeeding the current head up to *newNode->persistentNode* are guaranteed to be persistent. To prevent future enqueues from redundantly flushing these nodes, the enqueuer then sets *newNode*’s backward link to NULL (Line 189). Thus, each enqueue operation that reaches *newNode* from now on, during its backward walk, would not need to traverse the preceding Persistent nodes.

Like in the original MSQ, a concurrent enqueue might prevent the enqueue’s linking. In this case, the enqueuee tries to assist the obstructing enqueue and advance the tail to point to the node enqueuee by that obstructing enqueue (Line 191), before starting a new attempt to enqueue its own item.

C.2 The Dequeue Operation

The dequeue operation attempts to extract the oldest item, placed in the node subsequent to the dummy node. If the queue is empty when the dequeue operation takes effect, it returns NULL. But before returning, the failing dequeue must ensure that previous dequeues that emptied the queue survive a crash. It does so by copying the head’s index to its local head index and persisting it (Lines 140–141). Each thread’s local head index variable is placed in the thread’s cell in the *localData* array.

If the queue is not empty, the dequeuer attempts to advance the head by one node (Line 143), and on success – returns the oldest item to the caller. On failure it retries the whole scheme. Before returning, the dequeuer copies the new head’s index to its local head index and persists it (Lines 145–146), to comply with durable linearizability, which requires that completed operations be linearized.

A successful dequeue is responsible for reclaiming the dummy node recorded by the previous dequeue executed by the same thread. Before reclaiming, it must ensure that the node is unreachable by future operations. To make it unreachable by backward walks (of enqueue operations that will try to identify a non-persisted suffix), the dequeuer disconnects the backward link from the new dummy head to the previous one (Line 147). It then returns the Persistent and Volatile objects of the previous dummy node to the memory manager (Lines 148–150), and keeps a record of the current dummy node for its future reclamation (Line 151).

C.3 Recovery

The recovery procedure of OptLinkedQ resurrects all nodes reachable through backward links from the abstract tail until the node succeeding the dummy head. It then allocates matching Volatile objects and sets their forward links to form the linked list that constitutes the volatile queue. This is implemented as follows.

Let *headIndex* be the maximal index among the local head indices of all threads. The recovery does not modify these per-thread indices. It sorts all per-thread *lastEnqueues*’s indices that are valid (namely, their valid bit value matches the valid bit value of the associated pointer), bigger than *headIndex* and have an associated non-NULL pointer from largest to smallest, and gathers them with their matching per-thread last enqueue pointers to a single list of potential tails. The recovery then attempts to start a backward walk from each potential pointer, one after another. For each attempted pointer, if the index in the Persistent object it points to is different from the associated index kept in the appropriate *lastEnqueues* cell, or if a nonconsecutive index is encountered during the backward walk from it to the Persistent object with index *headIndex+1* (each of these cases implies that the index of the inspected Persistent object is stale) – the recovery moves on to try the next potential tail.

All Persistent objects in the designated allocation areas but the ones traversed in the last successful walk are reclaimed (if there

was such a walk, otherwise the queue is empty and all Persistent objects are reclaimed). For each reclaimed node with an index bigger than *headIndex* (there could be at most one such node per thread – for threads that were in the middle of enqueueing when the crash occurred), the recovery zeroes the node’s index and flushes it before retiring the node.

In order to construct a linked list of Volatile objects, for each of the recovered Persistent objects, the recovery allocates a Volatile object and sets its Persistent pointer to the associated Persistent object. In addition, the *index* and *item* of each Volatile are copied from the associated Persistent. The *next* pointers of the Volatile objects are set according to the queue’s order. The *prev* field of the last Volatile object is set to NULL. Dummy Volatile and Persistent objects are allocated too. Their *index* fields are set to *headIndex*. The Persistent pointer of the dummy Volatile object is pointed at the dummy Persistent object. The *next* pointer of the dummy Volatile is pointed at the recovered Volatile object with index *headIndex+1*, or set to NULL if an empty queue is recovered. The queue’s head and tail pointers are pointed at the first and last Volatile objects in the linked list respectively.

For all threads that do not contain a valid record of the recovered tail in any of their *lastEnqueues* cells, these cells are zeroed using *movnti* instructions. In addition, their *lastEnqueuesIndex* is set to 0, and their *validBit* is set to 1. For a thread with a valid *lastEnqueues* cell referring to the recovered tail: Its other cell is zeroed using *movnti* instructions. In addition, its *lastEnqueuesIndex* is set to the other cell’s index, and the thread’s *validBit* is set appropriately (so that the next write to the cell that refers to the recovered tail will be with a valid bit value opposite of its current one).

Finally, the recovery issues an SFENCE to ensure the completion of all executed flushes and *movnti* instructions.

D LOCK-FREEDOM PROOF

To prove lock-freedom in the presence of crashes, we need to prove that each time a thread executes an operation on the queue, and there are no interrupting crash events since the operation’s invocation, some thread completes an operation on the queue within a finite number of steps. Namely, it is sufficient to prove progress for crash-free intervals of execution. For each of the four described queue algorithms, the following holds: within $n+1$ loop iterations of a given running operation (assuming a crash-free long-enough interval of execution), where n is the number of threads operating on the queue, some operation succeeds to perform a linearization point.

We complete the argument brought in Section 8. We start with noting that an obstructing volatile linearization point of some operation does not cause another operation to branch backwards more than once: a dequeue obstructing another dequeue has advanced the head, so the interrupted dequeue will read a new value from the queue’s head in its next iteration, and an enqueue interrupted by another enqueue ensures the tail is advanced before starting a new iteration.

Next, we explain why in case of a dequeue operation, n iterations are sufficient to guarantee progress. Let the examined running *op* be a dequeue. It branches backwards each time another dequeue precedes it with advancing the head. If *op* does not complete within

n iterations, some other thread must have advanced the head twice, in two different dequeue operations. This means it must have completed the first dequeue operation of the two, denoted *firstDeq*. Prior to completing *firstDeq*, the other thread has persisted the head. Thus, *firstDeq* is linearized. We still need to show that the linearization point occurs within the n inspected iterations of op and not prior to them, in order to show that n iterations of a dequeue are enough to achieve progress. *firstDeq*'s linearization point occurs in op 's iteration in which *firstDeq* has failed op , because in this iteration op read the queue's head, and then failed to advance it since the obstructing *firstDeq* has advanced it in between.

For an enqueue, n iterations are not adequate to ensure progress. Let the examined running op be an enqueue. We analyze its execution since an iteration it started at moment t . op branches backwards each time another enqueue precedes it with linking a node to the tail. If a linearized enqueue fails op 's first linking attempt, it is not guaranteed that the linearization point of this enqueue occurs after

t . But from op 's second iteration on, each enqueue that fails op and is linearized – is guaranteed to be linearized after t : it is linearized when it links its node to a previous node denoted N , after the tail is advanced to point to N , which happens after op obtains the tail in the first inspected iteration (since its obtained value must point to a preceding node, to which another enqueue operation has linked a node). Therefore, we do not look at the first n iterations of op , but rather at the n iterations starting with the second one. A similar argument to the one brought for a dequeue op applies to these iterations: If op does not complete within $n+1$ iterations, some other thread must have linked twice within iterations 2 to $n+1$, in two different enqueue operations. This means it has completed the first enqueue of the two. Prior to returning from this enqueue, it has ensured the survival point of that enqueue. Thus, this enqueue is linearized. As explained before, its linearization point occurs after t , namely, within the $n+1$ inspected iterations of op .