A Persistent Lock-Free Queue for Non-Volatile Memory

Michal Friedman
Technion, Israel
michal.f@cs.technion.ac.il

Maurice Herlihy
Brown University, USA
mph@cs.brown.edu

Virendra Marathe
Oracle Labs, USA
virendra.marathe@oracle.com

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

Abstract

Non-volatile memory is expected to coexist with (or even displace) volatile DRAM for main memory in upcoming architectures. This has led to increasing interest in the problem of designing and specifying *durable* data structures that can recover from system crashes. Data structures may be designed to satisfy stricter or weaker durability guarantees to provide a balance between the strength of the provided guarantees and performance overhead. This paper proposes three novel implementations of a concurrent lock-free queue. These implementations illustrate algorithmic challenges in building persistent lock-free data structures with different levels of durability guarantees. In presenting these challenges, the proposed algorithmic designs, and the different durability guarantees, we hope to shed light on ways to build a wide variety of durable data structures. We implemented the various designs and compared their performance overhead to a simple queue design for standard (volatile) memory.

CCS Concepts • **Computing methodologies** Shared **memory algorithms**; **Concurrent algorithms**;

Keywords Non-volatile Memory, Concurrent Data Structures, Non-blocking, Lock-free

ACM Reference Format:
Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *PPoPP '18: PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3178487.3178490

This work was supported by the United States - Israel Binational Science Foundation (BSF) grant No. 2012171. Maurice Herlihy was supported by NSF grant 1331141.

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02. . . $15.00
https://doi.org/10.1145/3178487.3178490

1 Introduction

Memory is said to be *non-volatile* if it does not lose its contents after a system crash. Non-volatile memory is soon expected to co-exist with or even displace volatile DRAM for main memory (but not caches or registers) in many architectures. This has led to increasing interest in the problem of designing and specifying *durable* data structures, that is, data structures whose state can be recovered after a system crash.

A major challenge in designing durable data structures is that caches and registers are expected to remain volatile. Thus, the state of main memory following a crash may be inconsistent, missing all previous writes to the data structure that were present in the cache but not yet written into the main memory. Dealing with arbitrary missing words after a crash requires non-trivial data structure algorithms. These algorithms must guarantee that key data does get written to main memory (without incurring too much overhead), thus making it possible to restore the data structure to a consistent state.

It would be interesting to know whether libraries of highly optimized, high performance persistent data structures [21] can be built using ad hoc techniques informed by the data structure architecture and semantics. Previous work focuses solely on B-tree implementations [5, 6, 25, 32]. The interest in B-trees is natural given their prevalence in file system and database implementations. However, other foundational data structures are also used in application domains that care about persistence, e.g., hash tables in key-value stores [11, 29, 30] and persistent message queues [28, 31, 34]. Since traditional storage media have been block-based, all these applications persist these data structures by marshaling them to a block-based format. Doing so involves non-trivial overhead that was dwarfed by the high cost of disk access. As a result, the in-memory representation and on-disk (-SSD) representation of these data structures are quite different. Byte-addressable persistent memory can be used to create a unified persistent representation. As far as we know, no previous work attempts to build highly concurrent, *nonblocking* data structures, optimized for persistent memory.

In order to strive for high-performance, crash-resilient software on non-volatile memories, we propose to look at modern, highly-concurrent data structures, such as the ones used in java.util.concurrent, and enhance them to work with non-volatile memories. Designing such concurrent data structures for upcoming non-volatile memories requires meeting the combined challenge of high concurrency and non-volatile durability.

We study these challenges by designing a durable version of the lock-free concurrent queue data structure of Michael and Scott [24], which also serves as the base algorithm for the queue in java.util.concurrent. This concurrent data structure is complicated enough to demonstrate the challenges raised by concurrent durable data structures, and simple enough to demonstrate solutions. Careful thought is needed to define what it *should mean* for a concurrent structure to be correct and durable. Can the effects of operations in progress at the time of a crash be lost? What about operations that completed before the crash, or dependencies that span multiple structures (e.g., popping an element from one structure and pushing it into another)?

In the absence of durability concerns, *linearizability* [15] is perhaps the most common correctness condition for concurrent objects: each operation appears to "take effect" instantaneously at some point between its invocation and response. Linearizability is attractive because (unlike, say, sequential consistency) it is *compositional*: the joint execution of two (or more) linearizable data structures is itself linearizable.

Various definitions were proposed to formalize durability, e.g., [2, 9, 13, 17, 27]. In this paper we adopt and work with the definition of linearizable durability by Izraelevitz *et al.* [17]. Informally, durable linearizability guarantees that the state of a data structure following a crash reflects a consistent subhistory of the operations that actually occurred. This subhistory includes all operations that completed before the crash, and may or may not include operations in progress when the crash occurred. The main tool for achieving durable linearizability for a concurrent data structure is the use of explicit instructions that force volatile cached data to be written to non-volatile memory. While such *persistence barrier* instructions enforce correctness, they also carry a performance cost and their use should be minimized: forcing data from caches into non-volatile memory can take hundreds of cycles.

Durable linearizability is compositional: the composition of two durably linearizable objects is itself durably linearizable. Nevertheless, durable linearizability may be expensive, requiring frequent *persistence barriers*. An alternative, weaker condition is *buffered durable linearizability*. Informally, this condition guarantees that the state of the object following a crash reflects a consistent subhistory of the operations that actually occurred, but this subhistory *need not* include all operations that completed before the crash. Buffered durable linearizability is potentially more efficient than durable linearizability, because it does not require such frequent persistence fences. Unfortunately however, buffered durable linearizability is not compositional: the composition of two buffered durably linearizable data structures is not itself buffered durably linearizable.

To synchronize buffered durably linearizable objects, a specific linearizable operation, sync(), can be issued to force a single-object persistence barrier: a call to an object's sync() method renders durable all that object's operations completed before the call, although operations concurrently with the call may or may not be rendered durable. The sync() method is provided by the data structure and can be used to synchronize a safe (manual) execution of two (or more) data structures concurrently.

Our first contribution is the proposal of three novel designs of durable concurrent queues. It is easy to obtain a durable linearizable queue by adding many persistence barrier operations automatically. But, in general, the obtained performance can be very low. In this paper, we attempt to minimize the overhead and still achieve robustness to crashes. The first implementation, denoted *durable* queue, provides durable linearization. The second implementation, denoted *log* queue, provides durable linearization, as well as an additional property that we discuss next. The third implementation, denoted *relaxed* queue, provides buffered durable linearizability with an implementation of a sync() operation.

When crashes occur during an execution, it is often difficult to tell which operations were executed and which operations failed to execute. Durable linearizability guarantees the completion of all operations that were executed before a crash but does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. A persistent queue is not itself sufficient for program recovery. One needs the larger context, and the ability, upon recovery, to determine how much has been executed so far. Without the ability to distinguish completed operations from lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once.

In this paper we enable a more robust use of the queue, by defining a new (natural) notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. The *log* queue provides durable linearization and detectable execution. If the program that uses the queue follows a similar procedure for detecting execution, then it is possible to tell how much of the execution has completed on recovery from a crash, and program recovery at higher level becomes possible.

In the course of proving the proposed algorithms, we discovered an alternative definition of a stronger property satisfying durable linearizability that was easier for us to work with. We provide this definition, as well as a proof that it satisfies the original definition of durable linearizability in Section **??**.

The queue is a fundamental data structure that will find uses in future applications optimized for persistent memory. As mentioned above, several existing messaging systems use a FIFO queue at their core and could benefit from high-performance queue for non-volatile memories. For this study, we chose to extend Michael and Scott's queue due to its portability, simplicity and performance. There exist faster queues that employ the `fetch&add` instruction[26, 35], but they are not portable to platforms that do not support this instruction (e.g., SPARC), and they are also significantly more complicated.

We have implemented the three queue designs and measured their performance. As expected, implementations that provide durable linearization have a noticeable cost. Interestingly, however, implementations providing detectable execution do not add significant overhead over durable linearization and may be worthwhile in this case. Also as expected, implementations that provide only buffered durable linearizability obtain good performance when the sync() method is invoked infrequently.

The rest of this paper is organized as follows. Section 2 presents definitions, discusses the setting, and reviews previous work. Section **??** offers an alternative definition to previous works. Section 3 provides an overview of the three queue versions. We provide the details of the durable, log and relaxed queues in Sections 4, 5, and 6 respectively. A mechanism for memory management is presented in Section 7. The experimental evaluation for all three queues is presented in Section 8. Section 9 discusses related work and Section 11 concludes.A correctness proof for the durable queue is presented in Section 10.

2 Preliminaries

In this section we present some definitions and recall relevant previous work.

2.1 Execution and Durability

We extend the standard notion of execution to also reflect the transfer of data from the cache to memory.

Definition 2.1. *NVM view.* An NVM view at time t is the content of the non-volatile memory at time t.

The NVM content consists of data that resides on the non-volatile memory and persists through a crash.

Definition 2.2. *Configuration.* A configuration is an instantaneous snapshot of the system describing the value of all local and shared variables as well as the program counter of each thread. In adition, the configuration would describe for each variable its value in the cache (if it exists) and its value in the NVM view.

In our algorithms, we consider flush instructions that flush the content of a cache line to the NVM. A flush can also occur implicitly by hardware executing a cache line eviction. To simplify the analysis of our algorithms, it is useful to explicitly include in the execution the transfer of data from the cache to the memory.

Definition 2.3. *(Extended) Computation step.* A computation step is a thread's local step that reads or writes a thread's own local variables, a shared-memory access that accesses the shared objects, an invocation of a method, or a response. In addition to standard execution steps that access local or shared variables, we also explicitly consider a flush of a cache line from the cache to the NVM as a step in the execution. This step can be triggered explicitly by a specific thread executing a flush or implicitly by the hardware that evicts cache lines. We assume each step is atomic. Crashes may be considered as steps as well, which are done by the hardware.

We assume that writes of threads are to volatile cache. A write to address A appears in NVM only after its cache line is flushed to the NVM (in the execution).

Definition 2.4. *(Extended) Execution.* An execution consists of an alternating sequence of configurations and computation steps, starting with the initial configuration where the shared variables are initialized with predetermined initial values in the NVM. An execution is legal if:

1. Every thread follows its algorithm in the subsequence consisting of the steps it performs.
2. Every shared object behaves according to its sequential specification in the subsequence of steps that access it.
3. The NVM content in each configuration contains exactly its initial values updated by all previous flush steps in the execution so far.

Note that, as in Izraelevitz *et al.* [17], crashes are considered legal steps and the crash events partition an execution as $E = E_0C_1E_1C_2...E_{c-1}C_cE_c$, where c is the number of crash events in E. C_i denotes the i-th crash event, and $ops(E)$ denotes the sub-execution containing all events other than crashes. Note that $ops(E_i) = E_i$ for all $0 <= i <= c$. Following to Izraelevitz *et al.* [17], we call the sub-execution of E_i the i-th era of E.

In the rest of this paper, whenever we mention an execution, we refer to this extended notion.

2.2 Durable Linearizability

We start by recalling the standard notations required to define linearizability and the extension to durable linearizability.

Threads perform one method call at a time. A method call consists of two events: an *invocation* event is followed by a *response* event. An execution of a concurrent system is modeled by a *history*, a finite sequence of method invocations, responses and crash events. A *subhistory* of a history H is a subsequence of the events of H. We write a method invocation as $\langle x\, m\, A \rangle$, where x is an object, m a method (and the arguments), and A is a thread. We write a method response as $\langle x\, t\, A \rangle$, where t is a value (possibly *void*) or an exception.

A response *matches* an invocation if they have the same object and thread. A *method call* in a history H is a pair consisting of an invocation and the next matching response. For a method call m, its delimiting events are denoted $inv(m)$ and $res(m)$. An invocation is *pending* in H if no matching response follows the invocation. $complete operation$ in H is an operation that has a matching response follows the invocation. $complete(H)$ is the extension of H with all matching responses to pending invocations appended in the end. We use $trunc(H)$, to denote the set of histories that can be generated from H by removing some of the pending invocations.

For a thread A, the *thread subhistory*, $H|A$ is the subsequence of events in H whose thread names are A. For an object x, the *object subhistory* $H|x$ is similarly defined. Histories H and H' are *equivalent* if for every thread A, $H|A = H'|A$. A thread or object history is *well-formed* if every response has an earlier matching invocation.

A method call m_0 *precedes* a method call m_1 in history H if m_0 finished before m_1 started: that is, m_0's response event occurs before m_1's invocation event in H. Precedence defines a *partial order* on the method calls of H: $m_0 \prec_H m_1$. A *consistent cut* of a history H is a subhistory $G \subseteq H$ such that, if m_1 is in G, and $m_0 \prec_H m_1$, then m_0 is also in G.

A history H is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. If S is a sequential history, then $\prec_S$ is a total order. A *sequential specification* for an object x is a prefix-closed set of sequential histories called the *legal* histories for x. A sequential history H is *legal* if each object subhistory $H|x$ is legal for x.

Definition 2.5. [15] *Linearizability.* For history H, and partial order $\prec$ extending $\prec_H$, H is *linearizable* if there exists a $complete(trunc(H))$, H', and there is a legal sequential history S, with no pending invocations, such that

L1 $complete(trunc(H))$ is equivalent to S, and
L2 if method call $m_0 \prec_{H'} m_1$, then $m_0 \prec_S m_1$ in S.

We refer to S as a $\prec$*-linearization* of H.

Definition 2.6. *Durable Linearizability.* An object is *durably linearizable* if, a crash and recovery that follows linearizable history H, leaves the object in a state reflecting a consistent cut H' of H such that

DL1 ops(H') is linearizable
DL2 every complete operation of H appears in H'.

Izraelevitz *et al.* [17] show that durable linearizability is compositional.

We now move to defining buffered durable linearizability. Let us stipulate that each object provides a sync() method, which allows a caller make sure that operations completed prior to the sync() are made persistent before operations that follow the sync().

Definition 2.7. [17] *Buffered Durable Linearizability.* An object is *buffered durably linearizable* if, a crash and recovery that follows linearizable history H, leaves the object in a state reflecting a consistent cut H' of H such that

BDL1 H' is linearizable
BDL2 every completed sync() operation of H appears in H'.

Buffered durable linearizability is not compositional. For example, suppose a thread dequeues value x from p, and then enqueues x on q, where p and q are distinct buffered durably linearizable FIFO queues. Following a crash, the thread might find two copies of x, one in each queue, while if the composition of the two queues were buffered durably linearizable, the recovering thread might find x in p but not q, or in q but not p, or in neither, but never in both.

2.3 Detectable Execution

A caller of a data structure operation often needs to be able to tell whether an operation has executed even when a crash occurs. Imagine an operation that adds money to a bank account, or an operation that places an order for a car. One would like to know that such operations execute exactly once. A typical execution scenario is one where a thread has a durable list of operations to execute and it executes these operations one by one. Upon recovery from a crash, the thread needs to know which of its data structure updates were executed. Providing a mechanism to determine whether an operation completed during a crash is therefore beneficial.

We say that a data structure provides *detectable execution* if it provides a mechanism that, upon recovery from a crash, makes it possible to tell whether each operation executed while the crash occurred was completed or aborted.

In practice, an implementation of such a mechanism can take the following form. An announcement array (similar to [14]) will hold an entry for each thread in which the thread announces an intention to execute an operation, and provides space for a recovery mechanism to write the operation result. Upon recovery from a crash, the recovery process will set a flag in each such entry specifying whether the intended operation was completed, and deliver the result, if relevant. The log queue, proposed in this paper, provides detectable execution.

2.4 Hardware Instructions for Persistence

In the algorithms presented in this paper, we use a FLUSH instruction that receives a memory address and flushes the content of this address (together with its entire cache line) to the memory, making it persistent. On an Intel platform this translates to two instructions: CLFLUSH, SFENCE. It has recently been shown that CLFLUSH has store semantics as far as memory consistency is concerned (see page 710 of [16]), which guarantees that no previous stores will be executed after the CLFLUSH execution. The SFENCE instruction guarantees that the CLFLUSH instruction is globally visible before any store instruction that follows the SFENCE instruction in program order becomes globally visible.

2.5 The MS Queue

Our constructions extend Michael and Scott's queue [24] (denoted the MS queue). As explained in the introduction, we chose this queue because it is highly portable (it only uses the CAS instruction), it is used in the Java concurrency library, and it is adequately complex for a study of durable data structures.

The MS queue is built on an underlying linked-list in which references to the head and tail are held, called, respectively `head` and `tail`. The head points to a dummy node that is not considered part of the queue, and is only there to allow easy handling of the empty list. The list is initiated to a single (dummy) node referenced by both the head and the tail.

To dequeue an element, the dequeuing thread tries to move the `head` to point to `head->next` using an atomic CAS instruction. Upon success, it retrieves the value in the new node pointed to by the head and upon failure it begins from scratch.

To enqueue an element, a new node is allocated and initialized with the required value and a null next pointer. If `tail->next` is NULL, then the enqueue attempts to let `tail->next` point to its node using an atomic CAS instruction. Upon success, it then moves `tail` to point to `tail->next`. Upon failure to append the node, the operation goes back to inspecting the tail and attempting to append at the end. If `tail->next` is not NULL, this means that the previous operation has not completed and the tail must be fixed to point to the last node. Thus, the current thread attempts to fix the tail and then it starts from scratch.

The linearization point of a dequeue operation is at a successful CAS executed on the head. Enqueuing an element is linearized in a successful CAS appending a node at the end of the queue. Note that the tail can later be fixed by the thread performing the enqueue or by any other enqueue operation that needs to append its node at the end. No appending is attempted before the tail is fixed and pointing to the last node in the queue. A full description of this queue appears in the original paper [24].

3 An Overview of the Three Queue Designs

Our queue builds on Michael and Scott's queue (denoted the *MS queue*) and consists of a linked list of nodes that hold the enqueued values, plus the `head` and the `tail` references. The basic original queue is extended with FLUSH operations to persist memory content required for recovery from crashes, and also with additional information that facilitates recovery.

Our three designs offer varying levels of durable linearization with guarantees provided to the caller. The *durable* version provides durable linearizability, the *log* queue provides both durable linearization and detectable execution, and the *relaxed* version provides buffered durable linearizability.

3.1 The Durable Queue

The durable queue satisfies durable linearizability, implying that any operation that completes before a crash must become persistent after the recovery. The first guideline we use in constructing this queue is the *completion guideline*, which states that when an operation completes, its effect is durable. This ensures that when a crash occurs, previously completed operations are bound to persist. In addition, a dependence order must be maintained in this construction between all operations that occur concurrently to a crash. For example, if two dequeues occur concurrently with a crash, linearizability dictates that if the second dequeue completes (the one that dequeued the later value in the queue), then the earlier dequeue must complete as well. Ensuring this requires extra care. Therefore, the second guideline we use is the *dependence guideline* by which each operation must ensure that all *previous* operations become durable before starting to execute. *Previous* here refers to operations that the current operation depends on and that must be linearized before the current operation can be linearized. We recommend this guideline be followed for all future constructions. Finally, we use a third and generally recommended *initialization guideline*, by which all fields of an object are flushed after the object is initialized and before the object is added to (i.e., before it becomes reachable from) the data structure. An overview on how the above three guidelines are implemented for the queue follows.

The enqueue operation of the durable queue starts by allocating a node and initializing it with the enqueued value and with a NULL next pointer. Next, it FLUSHes the node content to memory. This ensures that, before this node is appended to the queue, its durable content becomes updated. Next, recall that the original MS enqueuer attempts to append the node to the end of the queue and then fix the tail to point to the appended node. Appending is only allowed if the tail points to the last node, whose next pointer is NULL. If this is not the case, the enqueuer first fixes the tail and only then tries again to append its own node.

In the extended enqueue of the durable queue, we add a FLUSH instruction after appending the new node and before fixing the tail. This FLUSH persists the pointer from the previous last node to the newly appended node. This flush satisfies the first guideline: at this point the operation is durable. If a crash occurs, the new node is safely persistent in the list.

Next, we turn to satisfying the second guideline. We need to make sure that a previous enqueue is made persistent before a new enqueue operation starts. That should be done when one thread adds a node at the end but pauses before flushing the pointer to the added node (and also before fixing the tail). In this case, extra care should be taken when helping to fix the tail for another operation. If an enqueuer needs to fix the tail following an incomplete previous enqueue operation, then it also flushes this pointer (that links the added node to the queue) before fixing the tail to point to this node.

The flush operations described above ensure that after enqueuing a node, the node content is durable and the pointer leading to it from the linked list is durable. Namely, all the backbone pointers of the linked-list underlying the queue are durable, except possibly the last updated pointer, whose enqueuing operation has not yet completed. The `tail`, on the other hand, need not be durable. During a recovery we can find its value by chasing the linked-list from the current location of the `head` until the last reachable node.

To make the `dequeue` operation durably linearizable, we need to add more than just flushes. First, we add a node field `deqThreadID`. This field in the queue node points to the thread that dequeued (the value in) this node. The `deqThreadID` field serves two purposes. First, it provides a direction for the recovery procedure to place the dequeued value at the disposal of the adequate dequeuer if a crash occurs. To facilitate such a recovery, we keep a `returnValue[]` array with an entry for each thread, in which a returned value can be placed. Second, it allows one dequeue operation to ensure that a previous dequeue operation completes and is made persistent.

To dequeue a value of a node, a dequeuer attempts to write its thread ID in the `deqThreadID` field of node `head->next` using an atomic CAS instruction. Whether successful or not, it then flushes the `deqThreadID` field to the memory to make sure that the thread which succeeded in this dequeue is recorded in the NVM. Next, it places the node's value in `returnedValues[deqThreadID]`, flushes this field to the memory to make sure the result is durably delivered to the caller, and updates the head to point to the next node. If the dequeuer did not manage to write its own thread ID into `deqThreadID`, then (after helping) it starts again by trying once more to place its thread ID in the `deqThreadID` field of `head->next`.

These operations provide durability as required. When an operation completes, its effects are persistent, and before an operation starts, it makes the effects of the previous operation persistent. However, this design has performance costs due to the added FLUSHes. Measurements of this cost are given in Section 8.

To recover from a crash, we fix the head, making sure that all dequeued values are placed in their intended locations, and place the head over the last node whose `deqThreadID` is non-NULL. We then also fix the tail to point to the last reachable node in durable memory.

The full algorithmic details appear in Section 4.

3.2 The Log Queue

The second implementation provides durable linearization and also detectable execution. This means that following a recovery after a crash, each thread can tell whether its operation has been executed, and it receives the results of completed operations. The *log* queue implementation employs a log array for the threads. An operation starts by being announced on a thread log. An operation is assign an operation number that is given by the user invoking thread such that the thread ID and the operation number uniquely identify the operation. The log contains the operation number, a flag that signifies if the operation completed, and an additional field that holds the operation result. If a crash occurs, then it is possible to simply inspect the log entry that contains the relevant operation number after the recovery to determine whether an operation of a crashed thread was executed or needs to be started again. Thus, the program can execute each of its intended operations exactly once.

Our general methodology for combining durability with detectable execution is to start with the durable version of the data structure and extend it with a mechanism to notify that the operation has completed. We demonstrate this approach on the queue. In the log array we maintain, for each thread, a log object on which a thread announces its intent to execute an operation, and on which the result and the operation numbers are written. The algorithmic details appear in Section 6. We provide an overview next.

The `enqueue` operation of the log queue starts by allocating a log object and a new queue node. Both the log object and the queue node are first initialized. The node's `value` is determined by the input, the node's `next` field is set to NULL, and the node `logInsert` field points to the log object. The log object is initialized with a pointer to the new node, with an indication that the operation is enqueued, and with an operation number that is assigned by the invoking thread. The contents of the node and the log object are then flushed to memory. Next, a pointer to this log object is placed in the log array (at the entry of the enqueuer thread) and this array entry is also flushed. Next we try to append the new node at the end of the queue and, if successful, we flush the appending pointer, namely, from the previous last node to the current last node, which has just been appended. Finally, we update the tail. No flushing is required for tail updates.

If the tail is not pointing to the last node, we need to fix the tail. Before fixing the tail, we flush the last pointer and fix the tail. After fixing the tail it is possible to try again to append our node at the end of the queue.

To `dequeue` a node we start by allocating a log object and initialize it to indicate the dequeue operation and the operation number. The log object is then flushed, a pointer to it is placed in the log array, and this entry in the log array is flushed as well. We then try to write a pointer to the log object into the `logRemove` entry of node `head->next`. Upon success, we flush the content of the `logRemove` field. We then put a pointer in the log to this node (which indicates that the operation has completed) and flush the log content as well. Finally, we advance the head to `head->next`. (The `head` need not be flushed.) A thread that fails to write its log entry into node `head->next` helps complete the dequeue operation (including flushing, linking to the log, flushing, and updating the head) and then tries its operation again.

During recovery, we start from the head and walk the linked list. Whenever we see a pointer to a log, we check whether the operation is completed, and if not, we complete the operation. The head is set to the last node that has a non-NULL `logRemove` field. We then proceed to update the tail to the last element in the list. We also make sure the last enqueue is completed by following the above procedure for marking the completion of the enqueue operation in the relevant log before fixing the tail for the last time. Finally, we go over all log entries and complete all the unfinished operations.

The proposed algorithm inherits from the durable queue the completion, dependence, and initialization guidelines for all of the operations included there. We use the initialization guideline for initializing the log object, while the dependence guideline ensures that previous operations become durable before we execute our own. In addition, we use a *logging guideline*, which ensures that the log with the description of the intended operation and the operation number is flushed before the operation is executed. This in turn ensures that, upon recovery, the operation will be completed.

3.3 The Relaxed Queue

The *relaxed queue* implementation provides buffered durable linearization, which is a weaker requirement. Buffered durable linearization only mandates that, upon failure, a proper prefix of the linearized operations take effect after recovery, while the rest of the operations are lost. There is no need to recover all operations that completed before the crash and thus no need to make an operation durable before returning. Hence, we adopt different guidelines, to maximize performance. The implementation needs to provide a sync() method that forces previous operations to become durable before later operations become durable. Typically, a caller invokes the sync() method to ensure proper compositionality between different data structures; occasionally, it does so to make sure not too many operations are lost when a crash occurs.

We use a design pattern that can be used for other data structures as well. During the execution of a sync() operation, we obviously make all previously executed operations durable, but we also save the state of the queue. In case of a crash, we (boldly) discard all operations that followed the last sync(), by returning to the saved state from the latest sync(). This may seem a painful loss of operations during a crash, but it efficiently satisfies the (weak) requirement of buffer durability and it allows the queue to be saved at different frequencies. The algorithm can completely avoid executing any FLUSH instructions inside the enqueue and dequeue operations. We only execute FLUSHes in the sync() method. This implies low overhead if crashes and sync() invocations are infrequent. Buffered durable linearizability is guaranteed because a consistent cut (a proper prefix) of the executed operations is always recovered after a failure. We call this design pattern *return-to-sync*.

To apply this idea to saving a state of the queue, we first note that nodes in the queue are essentially immutable from the moment they are appended to it. So if we look at a current queue state and would like to elide several recent operations and return in time to an earlier state, it suffices to simply restore `head` and `tail` to their previous values at that earlier time (and set `tail->next` to NULL). Keeping this in mind, we add to the queue state two variables, `saved_head` and `saved_tail`, which hold the values of `head` and `tail` the last time sync() was called. Whenever a crash occurs, we can set `head` and `tail` back to their saved values. For this to work properly, we need to make all the nodes between `saved_head` and `saved_tail` persistent. The sync() method ensures this by performing the required flushes.

The above motivating discussion implies what the sync() method should do. This method starts by reading the current `head` and `tail` values. It then flushes the content of all nodes in between these two pointers to the durable memory, and finally, it attempts to replace the previously saved `head` and `tail` with the current ones. The first challenge is to obtain an atomic view of `head` and `tail`, in order to make sure that a consistent cut (i.e., a proper prefix) of the operations is made persistent. A second challenge is to replace the values of `saved_head` and `saved_tail` simultaneously. The third challenge is to coordinate multiple sync() operations and make sure that the most updated consistent cut is saved to NVM.

We solve the first challenge by marking the `tail` pointer, after which it does not change until the `head` and `tail` are saved. It is important that we not mark the `tail` in the middle of an enqueue operation. This can be enforced by helping to complete previous operations. The simultaneity challenge is simply solved by holding `saved_head` and `saved_tail` inside an object that is replaced by a single CAS instruction. The third challenge is solved by obtaining a global number that indicates the order of the sync() operations and dealing with races that come up. The full algorithmic details appear in Section 6. For the relaxed queue, the completion guideline is irrelevant. The return-to-sync design pattern makes the dependence and the initialization guidelines irrelevant as well.

4 Algorithm Details of the Durable Queue

As mentioned above, our queue extends the MS queue. Its underlying data structure includes a linked-list of queue nodes and the `head` and `tail` pointers. The first node in the linked-list is a sentinel node that allows simple treatment of an empty list. The implementations use FLUSH in order to maintain different levels of guarantees. The FLUSH operation consists of two hardware instructions, as discussed in Section 2.4. In this section, we provide the details of the `durable` queue. Our queue's underlying representation is a singly-linked list

```
1  class Node {
2      T value;
3      Node* next;
4      int deqThreadID;
5      Node(T val) : value(val), next(NULL), deqThreadID(−1) {}
6  };
7
8  class DurableQueue {
9      Node* head;
10     Node* tail;
11     T* returnedValues[MAX_THREADS];
12     DurableQueue() {
13         T* node = new Node(T());
14         FLUSH(node);
15         head = node;
16         FLUSH(&head);
17         tail = node;
18         FLUSH(&tail);
19         returnedValues[i] = NULL; // for every thread
20         FLUSH(&returnedValues[i]);
21     }
22 };
```

Figure 1. Internal Durable Queue classes

with a sentinel node. It builds on the inner `Node` class, which holds elements of the queue's linked-list. In addition to the standard node fields, i.e., the value and the pointer to the next element, the node class also contains an additional field (line 4): *deqThreadID*. This field holds the ID of the thread that removes the node from the queue.

The durable queue class contains two pointers and an array. The pointers *head* and *tail* point to the first and last nodes of the linked list that implements the queue. The *returnedValues* array is an array of pointers to objects that hold dequeued values (line 11). This array contains an entry for each thread and its size is MAX-THREADS, which is the number of threads that might perform operations on the queue.

The *returnedValues* array entries point to an object that contains a single `value` field. This field either contains a value that has been dequeued from the queue, or one of three special values that are not valid queue values:

1. The special *NULL* value signifies that the thread is currently idle (this is the initial value).
2. The special *pending* value indicates the intention of the thread to remove a node.
3. The special *empty* value is returned when the queue is empty.

The queue constructor initializes the underlying linked list with one sentinel node. It lets the head and the tail point to this sentinel node, and it also initializes the returned values array with the special NULL value. In order to persist these values, we flush the sentinel node, the head and the tail pointers, and the *returnedValues* array.

```
1  void enq(T value) {
2      Node* node = new Node(value);
3      FLUSH(node);
4      while (true) {
5          Node* last = tail;
6          Node* next = last−>next;
7          if (last == tail) {
8              if (next == NULL) {
9                  if (CAS(&last−>next, next, node)) {
10                     FLUSH(&last−>next);
11                     CAS(&tail, last, node);
12                     return;
13                 }
14             } else {
15                 FLUSH(&last−>next);
16                 CAS(&tail, last, next);
17             }
18         }
19     }
20 }
```

Figure 2. Enqueue operation of Durable Queue

Theorem 4.1. *The durable queue is durably linearizable.*

Correctness arguments for the queue, its progress guarantee and durability (proof of theorem 4.1) appear in Section 10

4.1 The Enqueue() Operation

The pseudo-code for the enqueue operation is provided in Figure 2. The *enqueue* method receives the value to be enqueued and it starts by creating a new node with the received value (line 2). It then flushes the node content (line 3). Next, the thread checks whether the tail refers to the last node in the linked list (lines 7-8). If so, it tries to append the new node after the last node of the list (line 9). Insertion consists of two actions: adding the new node after the last node and updating the tail to reference the newly added node. To ensure proper durability, we add a flush between the two actions. If insertion at the end is successful, we flush the next pointer of the previous node (line 10), and only then update the tail (line 11). Failure in updating the tail means that another thread has helped update it already completed the insertion of the current thread. Failure in the first CAS instruction (line 9) means that another thread has appended a different node and the operation starts from scratch (line 5). In case the next pointer of the last node does not point to NULL (line 14), we help complete the previous enqueue operation by flushing the previous next pointer (line 15) and fixing the tail (line 16).

The consistent flushing of the next pointer before updating the tail and the flushing of the node content after its initialization yield an important invariant: the entire linked list, up until the current tail, is guaranteed to reside in the volatile memory. This enables the correct execution of the recovery procedure.

```
1  void deq(int threadID) {
2      T* newReturnedValue = new T();
3      FLUSH(newReturnedValue);
4      returnedValues[threadID] = newReturnedValue;
5      FLUSH(&returnedValues[threadID]);
6      while (true) {
7          Node* first = head;
8          Node* last = tail;
9          Node* next = first−>next;
10         if (first == head) {
11             if (first == last) {
12                 if (next == NULL) {
13                     *returnedValues[threadID] = EMPTY;
14                     FLUSH(returnedValues[threadID]);
15                     return;
16                 }
17                 FLUSH(&last−>next);
18                 CAS(&tail, last, next);
19             } else {
20                 T value = next−>value;
21                 if (CAS(&next−>deqThreadID, −1, threadID)) {
22                     FLUSH(&first−>next−>deqThreadID);
23                     *returnedValues[threadID] = value;
24                     FLUSH(returnedValues[threadID]);
25                     CAS(&head, first, next);
26                     return;
27                 } else {
28                     T* address =
29                         returnedValues[next−>deqThreadID];
30                     if (head == first) { //same context
31                         FLUSH(&first−>next−>deqThreadID);
32                         *address = value;
33                         FLUSH(address);
34                         CAS(&head, first, next);
35                     }
36                 }
37             }
38         }
39     }
40 }
```

Figure 3. Dequeue operation of Durable Queue

4.2 The Dequeue() Operation

The dequeue operation receives a thread ID of the dequeuer. It starts by creating and initializing to NULL a new T object, whose purpose is to hold the dequeued value (line 2). Next, it flushes T's content (line 3). Then, it puts a reference to T in the `returnedValues` array and flushes the array entry (lines 4-5). Next, the thread checks that the queue is not empty and that the tail points to the last node (lines 7-18).

If the queue is empty, the method updates the corresponding entry in the `returnedValues` array with the empty value, flushes it and returns (lines 13-15). If the head and tail refer to the same node and the tail must be fixed, i.e., there is some enqueue operation in progress (line 11-12), then the dequeuer helps complete this enqueuing operation. It flushes the next pointer of the previous node (in our case the sentinel), fixes the tail and returns to the beginning of the while loop (lines 17-18). Dequeuing a node consists of following actions: marking the `deqThreadID` field of the node `head->next` with the dequeuer thread ID, writing the dequeued value in the `returnedValue` array, and promoting the head. If the thread succeeds in changing the `deqThreadID` field from NULL to his thread ID (line 21), then it flushes `deqThreadID` (line 22).

Next, it updates its entry in the array with the new value and flushes this result. Finally, it updates the head (line 25). Failure to update the head means that another thread has helped and completed the removal of the current thread. Failure to update the `deqThreadID` field means that another thread has already marked the node with its own threadID, so the dequeuer helps complete this other dequeue before starting again (lines 27-34). An important invariant here is that before the head is moved, the `deqThreadID` field content (which determines which thread receives the dequeued value) is made durable so that recovery can identify the winning dequeue operation. Also, before the head is advanced, the dequeued value is written to the `returnedValue` array and the returned value is flushed. Therefore, once the head advances in main memory, we know that the dequeuing of all previous nodes can be recovered.

4.3 The Recovery() Operation

After a crash, we assume that new thread start executing when the system recovers. There new threads execute the recovery method before starting to execute data structure operations. As a situation where some threads may finish the recovery operation before others, it was designed in a way that it can run concurrently to threads that execute data structure operations as well.

We divide the recovery procedure into four logical parts: Let A be the last node that has a non-NULL `deqThreadID` field, and let B be the last node that is reachable from the `head` and whose next pointer is NULL.

1. If A's predecessor is reachable from the `head`, then update the `head` by a CAS operation to point to the predecessor of A. Otherwise, skip to the third step.
2. Flush A's `deqThreadID` field, update the `returnedValues` array to point to the node's value , flush the array entry and update by a CAS operation the `head` to point to A (This becomes the new sentinel node).
3. If B's predecessor is reachable from the `head`, then update the `tail` by a CAS operation to point to the predecessor of B. Otherwise, update the `tail` to point B and skip the fourth step.
4. Flush the `next` pointer of the node the `tail` points to, and then update the `tail` by a CAS operation to point to B.

```
1  class Node {
2      T value;
3      Node* next;
4      LogEntry* logInsert;
5      LogEntry* logRemove;
6      Node(T val) : value(val), next(NULL),
7      logInsert(NULL), logRemove(NULL) {}
8  };
9
10 class LogEntry {
11     int operationNum;
12     Operation operation;
13     bool status;
14     Node* node;
15     LogEntry(bool s, NodeWithLog* n, Operation a, int opNum) :
16     operationNum(opNum), operation(a), status(s),
17     node(n) {}
18 };
19
20 class LogQueue {
21     Node* head;
22     Node* tail;
23     LogEntry* logs[MAX_THREADS];
24     LogQueue() {
25         T* sentinel = new Node(T());
26         FLUSH(sentinel);
27         head = sentinel;
28         FLUSH(&head);
29         tail = sentinel;
30         FLUSH(&tail);
31         logs[i] = NULL; // for every thread
32         FLUSH(&logs[i]); // for every thread
33     }
34 };
```

Figure 4. Internal Log Queue classes

Note that the head and the `tail` are not flushed during any run explicitly, so the recovery operation might go through nodes that was already removed and have a `deqThreadID` field which is not -1. In order to complete operations that were partially executed before the crash, we need to update the head, the `tail` and the last node that has a non-NULL `deqThreadID` field may have the array entry not updated. According to that, and the fact that threads that complete the recovery phase may tun concurrently to threads that are still in the recovery phase, we only promote the head and the `tail` to one node before the node they suppose to point to. Then, we do the operations that we would usually do during the normal run, which means the normal help operations, that are similar to the second and the fourth step in the recovery. It makes the procedure generic, that doesn't really has to differentiate between the nodes that have been changed during the previous and the current run.

5 Algorithm Details of the Log Queue

This section provides details of the *log* queue, which extends the durable queue and provides both durable linearization and detectable execution.

We start by adding two additional fields to the queue nodes (see the Node class (lines 4-5 in Figure 4). The `logInsert` field specifies which thread enqueued this node. More specifically, this field points to the log object that represents the intent to enqueue this node on the queue. The second `logRemove` field points to the log object of the dequeuer thread that dequeued this node. Initially these pointers are NULL. To construct logs, we also introduce an inner class named `LogEntry`. A LogEntry object holds details about an operation that a thread executes. The LogEntry class contains four fields: the `operationNum` field is a unique identifier for this operation set by the caller. The `operation` field contains either *enqueue* or *dequeue*. The `status` flag indicates whether the operation has been completed, if relevant. Finally, the `node` field, which points to the associated node (which needs to be enqueued, or has been dequeued). The log queue keeps a *logs* array to maintain a log. This array has an entry for each thread, it points to this thread's logEntry, and its size is MAX-THREADS, which is the number of threads that might perform operations on the queue.

The queue constructor initializes the underlying linked list with one sentinel node. It lets the head and the tail point to this sentinel node, and it also initializes the *logs* array with NULL values indicating that no operation is currently ongoing. Finally, it flushes all relevant content: the sentinel node, the head and tail pointers, and the content of the *logs* array.

5.1 The Enqueue() Operation

The pseudo-code for the enqueue operation is provided in Figure 5. The enqueue method works with a log entry describing the intended operation and a link to the inserted node. After initialization, the node is appended to the list and the log is updated with a flag indicating completion.

The enqueue method receives the value to be enqueued, the ID of the thread that performs the enqueue and the operation ID. It starts by allocating a new log entry to describe the intended operation. It is initialized with an *enqueue* operation, a *completed* status (FALSE), a pointer to a new node with the given value and the operationNumer that specifies the operation ID. Next, it sets the `log` pointer in the new node to point to the log entry. Then, the content of the node and the log are flushed. Next, the thread entry in the log array is set to point to this log entry, and the log array field is flushed. The actual insertion to the queue consists of two actions: appending the new node at the end of the list and updating the tail to reference the newly added node. To ensure proper durability, we add the following flush instructions.

If appending the node is successful, we flush the next pointer of the previous node (line 17). Only afterwards do we

```
1  void enq(T value, int threadID, int operationNumber) {
2      LogEntry* log = new LogEntry(false, NULL, enqueue,
3                                   operationNumber);
4      Node* node = new Node(value);
5      log−>node = node;
6      node−>logInsert = log;
7      FLUSH(node);
8      FLUSH(log);
9      logs[threadID] = log;
10     FLUSH(&logs[threadID]);
11     while (true) {
12         Node* last = tail;
13         Node* next = last−>next;
14         if (last == tail) {
15             if (next == NULL) {
16                 if (CAS(&last−>next, next, node)) {
17                     FLUSH(&last−>next);
18                     CAS(&tail, last, node);
19                     return;
20                 }
21             } else {
22                 FLUSH(&last−>next);
23                 CAS(&this−>tail, last, next);
24             }
25         }
26     }
27 }
```

Figure 5. Enqueue operation of Log Queue

update the tail (line 18). In this case we do not bother setting the status flag to true, because flushing the next pointer of the previous node indicates that the operation executed. Failure in the CAS operation to update the tail means that another thread has helped update it and has already completed the enqueue operation of the current thread. Failure in the first CAS instruction (line 16) means that another thread has appended a different node and the operation starts from scratch (line 11). If the next pointer of the last node does not point to NULL (line 21), we help complete the previous enqueue operation by flushing the previous next pointer (line 22). Finally, we fix the tail (line 23).

We maintain an important invariant that all the nodes in the queue, up to the current node pointed to by the tail, are persistent. In addition, all their log entries are fully updated in persistent memory. This ensures a consistent view for the recovery procedure. The last enqueue may not survive a crash, depending on whether the append of its node to the end of the queue persists. But it is easy to tell during the recovery whether a logged operation has completed.

5.2 The Dequeue() Operation

The pseudo-code for the dequeue operation is provided in Figure 6. The dequeue operation receives the ID of the thread that performs the dequeue. Then, it allocates a new log entry with a proper initialization and flushes the log content (lines 29-31).

```
28 void deq (int threadID, int operationNumber) {
29     LogEntry* log = new LogEntry(false, NULL, dequeue,
30                                  operationNumber);
31     FLUSH(log);
32     logs[threadID] = log;
33     FLUSH(&logs[threadID]);
34     while (true) {
35         Node* first = this−>head;
36         Node* last = this−>tail;
37         Node* next = first−>next;
38         if (first == this−>head) {
39             if (first == last) {
40                 if (next == NULL) {
41                     logs[threadID]−>status = true;
42                     FLUSH(&(logs[threadID]−>status));
43                     return;
44                 }
45                 FLUSH(&last−>next);
46                 CAS(&tail, last, next);
47             } else {
48                 if (CAS(&next−>logRemove, NULL, log)) {
49                     FLUSH(&first−>next−>logRemove);
50                     next−>logRemove−>node = first−>next;
51                     FLUSH(&first−>next−>logRemove−>node);
52                     CAS(&head, first, next);
53                     return;
54                 } else {
55                     if (head == first) { //same context
56                         FLUSH(&first−>next−>logRemove);
57                         next−>logRemove−>node = first−>next;
58                         FLUSH(&next−>logRemove−>node);
59                         CAS(&head, first, next);
60                     }
61                 }
62             }
63         }
64     }
65 }
```

Figure 6. Dequeue operation of Log Queue

Next, it lets its entry in the *logs* array point to this log persistently (lines 32-33). The actual dequeue starts then. If the queue is empty, the thread updates its corresponding log status as true. The NULL pointer to the node indicates an empty returned value (lines 41-43). If the queue is empty and there is an enqueue operation in progress, then the dequeuer attempts to help complete the enqueue operation and dequeue again (lines 45-46). If the queue is not empty, a dequeue consists of three actions: setting the `logRemove` field of `head->next` to point to the log entry of the dequeuer (using an atomic CAS), letting this log entry point to the dequeued node, and advancing the `head`. In this case we do not bother setting the status flag to true, because the node pointer indicates that the operation executed. If the first action (line 48) succeeds, the dequeuer flushes the logRemove field (line 49), updates the log entry, and flushes it as well. Only then does it update the head (line 52). Failure to update the head means that another thread has helped and completed the removal of the current thread. Failure to set the logRemove field of the node means that another thread has taken over this node, and we help that thread complete the dequeue before trying again. Here, we do not need to update the status because the association of the log with the node already indicates that the operation has been executed.

The main invariant here is that all dequeues, except maybe for the last one, are properly reflected in the log entries, and also have the dequeued node point to the appropriate log entry. The last dequeue is made durable only if the setting of the logRemove field persists after the crash. If the dequeue was not made durable, we will complete it after the crash.

5.3 The Recovery() Operation

Recall that each thread executes the recovery procedure after a crash and only then proceeds with its normal operations. The recovery procedure argument is the *logs* array from the NVM-view. The procedure can be divided into six logical parts. Let *A* be the last node that has a non-NULL logRemove field, and let *B* be the last node that is reachable from the head and whose next pointer is NULL (Note that the nodes referred to as *A* and *B* change dynamically with changes in the queue).

1. If *A*'s predecessor is reachable from the head, then update the head by a CAS operation to point to the predecessor of *A*. Otherwise, skip to the third step.
2. Flush *A*'s logRemove field, update the log entry that is pointed by the logRemove field, to point to this node, flush the array entry and use a CAS operation to update the head to point to *A* (This becomes the new sentinel node).
3. If *B*'s predecessor is reachable from the head, then update the tail by a CAS operation to point to the predecessor of *B*. Otherwise, update the tail to point *B* and skip to the fifth step. During the traversal, update the status field of the logInsert field of all the traversed nodes to be TRUE. This update is crucial for the fifth step, to avoid executing operations twice.
4. Flush the next pointer of the node that the tail points to, update the relevant status of the log to be TRUE. Then, update the tail by a CAS operation to point to *B*.
5. Traverse the *logs* array to finish all the started operations.
6. Create a new *logs* array and replace the old one by a CAS operation (if the old *logs* array has not been previously changed by another thread).

The main addition to the recovery function of the durable queue is the *logs* array traversal. We traverse the *logs* array and finish all the started operations.

For a dequeue operation, if the log entry is already referenced by the removed node, we flush the node field in the log entry and continue to the next log entry. If not, we sample the head and then recheck that the node field in the log entry is NULL. If so, we help the thread to finish its operation as follows. We check the head again, and if the head has been changed, we sample the node field in the current log entry again, and try again to finish the operation, unless it was already completed by another thread.

For an enqueue operation, we do something similar. If the log entry status flag is TRUE or the logRemove field is not NULL, we continue to the next log entry as this operation has completed. If not, we sample the tail and then recheck that the status field in the log entry is FALSE. If so, we help the thread to finish its operation as follows. We check the tail again, and if the tail has been changed, we sample the status field in the current log entry again, and try again to finish the operation, if this operation was not already completed. In this way, we first finish all pending operations from the previous phase, and only then, do we start to execute the operations from the current phase. We proceed to the current phase even if some threads are still running the recovery operation.

Note that if a certain thread finished its recovery phase and started running, threads that are in the middle of the recovery phase, will not execute any operation during that phase as there was already one thread that executed all those operations.

6 Algorithm Details of the Relaxed Queue

We now go into the details of the more efficient relaxed queue. This queue provides buffered durable linearization which is a weaker guarantee. The main idea is to maintain a snapshot of the queue, which has a consistent state by representing a prefix of linearized operations in the NVM view. Every sync() execution makes all the operations executed before the sync() persistent. This queue builds on two inner classes - Node and LatestNVMData. The Node class represents the nodes of the queue and is similar to the MS queue. The LatestNVMData class has three fields, NVMTail, NVMHead and a NVMVersion. It is used to keep record of the latest saved head and tail values. The RelaxedQueue class contains the standard *head* and *tail* and a LatestNVMData object named NVMState that maintains the latest persistent state of the queue. The invariant that we keep is that all nodes between NVMState->NVMTail and NVMState->NVMHead are durable, and so is the content of the object NVMState itself. An additional version field is used by the sync() method or other concurrent helping threads to distinguish one sync operation from another, so that older sync operations will not interfere with newer sync operations. In order to indicate that a snapshot is being recored by a sync operation, we use one additional class that inherits from Node.

```
class Node {
    T value;
    Node* next;
    Node(T val) : value(val), next(NULL) {}
};

class LatestNVMData {
    Node* NVMTail;
    Node* NVMHead;
    long NVMVersion;
};


class Temp: public Node {
    int version;
    Node* tail;
    Node* head;
    Temp(long v) : version(c), tail(NULL), head(NULL) {}
};

class RelaxedQueue {
    Node* head;
    Node* tail;
    LatestNVMData* NVMState;
    atomic<int> version;
    RelaxedQueue() {
        Node* sentinel = new Node(T());
        FLUSH(sentinel);
        head = tail = sentinel;
        FLUSH(&head);
        FLUSH(&tail);
        LatestNVMData* d = new LatestNVMData();
        d−>NVMTail = sentinel;
        d−>NVMHead = sentinel;
        d−>NVMVersion = −1;
        FLUSH(d);
        NVMState = d;
        FLUSH(&NVMState);
        version = ATOMIC_VAR_INIT(0);
    }
};
```

Figure 7. Internal Relaxed Queue classes

This class has the same fields as the `LatestNVMData` class and is called the `Temp` class.

The queue constructor initializes the underlying linked list with one sentinel node. It lets the head and the tail point to this sentinel node. It also initializes NVMState's `NVMTail` and `NVMHead` with pointers to the sentinel node, and `NVMVersion` with -1 value that indicates the initial state.

6.1 The Enqueue() Operation

The pseudo-code for the enqueue operation is provided in Figure 8. The enqueue method is similar to the MS queue. The only modification is in lines 13-17, where the thread checks whether it needs to help the sync() function. The

```
1  void enq(T value) {
2      Node* node = new Node(value);
3      while (true) {
4          Node* last = tail;
5          Node* next = last−>next;
6          if (last == tail) {
7              if (next == NULL) {
8                  if (CAS(&last−>next, next, node)) {
9                      CAS(&tail, last, node);
10                     return;
11                 }
12             } else {
13                 if (next == tempAddress) {
14                     CAS(&next−>head, NULL, head);
15                     CAS(&next−>tail−>next, next, NULL);
16                     continue;
17                 }
18                 CAS(&tail, last, next);
19             }
20         }
21     }
22 }
```

Figure 8. The enqueue method of the Relaxed Queue

sync() function records a snapshot of the head and the tail. To do that, it "freezes" enqueues and then records the value of the head and tail, knowing they reflect a snapshot. To freeze enqueues, a temporal node (that inherits from the original node) is set to the last node's `next` pointer. The condition in line 13 checks for this case, and helps the sync that is currently executing, by saving the current head to the current object (the tail is already set). It can then fix the last `next` pointer to NULL and restart the enqueue method execution.

6.2 The Dequeue() Operation

The pseudo-code for the dequeue operation is provided in Figure 9. The dequeue method is also similar to the MS queue with the exception that an alternative form of an empty queue is one where the sentinel's next pointer points to a temporal node. This indicates that the sync() method is currently running on an empty queue.

6.3 The Sync() Operation

The purpose of the sync() function is to make all the operations executed before the sync() persistent. To achieve that, we simply record a persistent snapshot of the queue state. Recovery can later return to this state and obtain a proper prefix (consistent cut) of the operations recovered. To record a persistent snapshot, we first save the values of `head` and `tail`, and then we make all node content between them persistent. The sync() function starts by allocating a new `Temp` object (with fields initialized to NULL) that will eventually replace the `NVMState` object if no other thread executes sync operations concurrently. To obtain an atomic view of `head`

```
T deq() {
    while (true) {
        Node* first = this−>head;
        Node* last = this−>tail;
        Node* next = first−>next;
        if (first == this−>head) {
            if (first == last) {
                if (next == NULL) {
                    throw EmptyException();
                }
                if (next == tempAddress) {
                    CAS(&next−>head, NULL, head);
                    CAS(&next−>tail−>next, next, NULL);
                    throw EmptyException();
                }
                CAS(&this−>tail, last, next);
            } else {
                T value = next−>value;
                if (CAS(&this−>head, first, next)) {
                    return value;
                }
            }
        }
    }
}
```

Figure 9. The dequeue method of the Relaxed Queue

and `tail` simultaneously, sync() blocks `tail` from changing (for a very short while) and then it reads both `head` and `tail`, and then unlocks the `tail`. To block `tail`, sync() installs a special pointer at the `next` field of the last node. This lets all other threads notice the block and help the sync() function before proceeding with further enqueue operations (lines 4-33). After sampling the `tail` and installing this special pointer, the sync() function (or any thread helping it) uses a CAS instruction to save a current `head` value (the `tail` value is already recorded in line 11) into the `Temp` object. Assuming a NULL value in that field, implies that the value of `head` in `Temp` can only change once and a stalled thread cannot later wake up and foil these values. Next, the special pointer in the last node can be switched back to NULL, enabling further enqueues to proceed (line 14). The sync() function proceeds by flushing the content of all nodes in the queue between the recorded `head` and `tail` (lines 36-42). Then it checks whether there was another snapshot that was concurrently recorded. A higher `NVMValue` value in the `NVMState` object represent a more progressed snapshot. Thus, if there was another snapshot that was concurrently recorded, with a higher value, then that snapshot can also be used by the currently executing thread, and we can simply exit. If another snapshot was recorded with a lower value, then it is outdated and cannot be used. In this case, we try to record the newer snapshot in `NVMState` and flush the `NVMState` field to the memory upon success. An optimization that we use is that if a thread notices a higher sync() operation

```
1  void sync() {
2      int currentVersion = 0;
3      Temp* temp = new Temp(currentVersion);
4      while (true) { }{
5          currentVersion = atomic_fetch_add(&version, 1);
6          temp−>version = currentVersion;
7          Node* last = tail;
8          Node* next = last−>next;
9          if (last == tail) {
10             if (next == NULL) {
11                 temp−>tail = last;
12                 if (CAS(&last−>next, next, temp) {
13                     CAS(&temp−>head, NULL, head);
14                     CAS(&last−>next, temp, NULL);
15                     break;
16                 }
17             } else {
18                 if (next == tempAddress) {
19                     if (next−>version > currentCounter ||
20                         next−>head == NULL) {
21                         CAS(&next−>head, NULL, head);
22                         CAS(&next−>tail−>next, next, NULL);
23                         temp = next;
24                         break;
25                     }
26                     CAS(&next−>head, NULL, head);
27                     CAS(&next−>tail−>next, next, NULL);
28                     continue;
29                 }
30                 CAS(&tail, last, next);
31             }
32         }
33     }
34     Node* currNode = temp−>head;
35     FLUSH(currNode);
36     while (true) {
37         if (currNode == temp−>tail)
38             break;
39         Node* next = currNode−>next;
40         FLUSH(next);
41         currNode = next;
42     }
43     LatestNVMData* currNVMState = NVMState;
44     LatestNVMData* potential = new LatestNVMData();
45     potential−>NVMTail = temp−>tail;
46     potential−>NVMHead = temp−>head;
47     potential−>NVMVersion = temp−>version;
48     while (true) {
49         if (currNVMState−>NVMVersion < temp−>version) {
50             FLUSH(potential);
51             if (CAS(&NVMState, currNVMState, potential)) {
52                 FLUSH(NVMState);
53                 break;
54             } else {
55                 currNVMState = NVMState;
56             }
57         } else {
58             break;
59         }
60     }
61     return;
62 }
```

Figure 10. The sync method of the Relaxed Queue

that is blocking the `tail` or a lower sync() function that is also blocking the `tail` but whose head is still not updated, then it can cooperate with that snapshot and use it. Another optimization that we use for large queues is that instead of flushing the content of all nodes in the queue between the recorded `head` and `tail`, we only flush the content between the previous recorded `tail` and the current recorded `tail`.

6.4 The Recovery() Operation

The recovery operation, simply sets the head and the tail to their saved values in the `NVMState` object. It also sets the `next` field of the node pointed to from the new tail to NULL. This can be done quickly by a single thread before all threads start running the operations on the queue.

Buffered durability is obtained here because, whenever a crash occurs, we simply return to a previous state of the queue. This means that a prefix of the previous operations takes place and a consistent cut of the operations remains.

7 Memory Management

Existing technology can be used to handle object reclamation. In particular, we can use hazard pointers [22] for reclamation and the Makalu NVM allocator [3] for allocations and reclamation after a crash. In our measured implementation we included hazard pointers, as described below, but did not incorporate the Makalu allocator. The Makalu allocator is new and we have not yet had the chance to examine it and use it. We simply used the standard G++ GNU allocator instead.

Makalu allocator reduces overhead by lazily persisting metadata that is easily recoverable if there is a crash, and by employing a garbage collector after a failure. In case of a crash, Makalu performs a parallel mark-sweep to reclaim all data that was not accounted for properly at crash time. It also restores certain metadata, which eliminates the necessity of ensuring full consistency of metadata at every allocation. The Makalu provides an API for allocating and freeing NVM objects, which can replace the `new` and `free` instructions in our implementation.

Memory management for lock-free data structures is a non-trivial task as threads may read a pointer to an object and then delay executing for a while. Until they wake up, the referenced object must not be reclaimed. Michael [22] proposed that threads announce objects that should not be claimed by publishing hazard pointers to them. In our implementation, we followed the hazard pointer scheme provided in [22] and used the implementation of [23]. Our extension had to ensure that hazard pointers are properly recorded also by threads that help other threads persist objects in the queue (in the durable and the log queues) and they were also used to enable reclamation of the log objects in these two queues. The most challenging work was with the relaxed queue as threads can concurrently call the sync() function and reclaim objects. We followed the principle that only a thread that managed to change the current snapshot was allowed to reclaim the objects from the head of the previous snapshot to the head of the current snapshot, and a thread that did not manage to update the snapshot once tried to take a new snapshot of the queue.

We did not attempt to persist the reclamation lists or hazard pointers. First, hazard pointers become irrelevant upon a crash, because threads do not proceed in the middle of an operation and so protecting objects that they locally reference is no longer required. New threads come up; they recover old operations and start new ones. As for the reclamation lists, they all contain objects that are no longer reachable from the data structure. Previously, they could only be reached from local pointers. After a crash, all objects not accessible from roots are reclaimed by an offline garbage collector that Makalu executes during the recovery.

Our implementation includes the hazard pointers and the measurements include the (non-trivial) overhead associated with maintaining them.

8 Measurements

We evaluated the performance of the proposed three queue implementations by comparing them one against the other and also against the original MS queue. We ran measurements on a 64-core machine, featuring 4 **AMD** Opteron(TM) 6376 2.3GHz processors, each with 16 cores. The machine has 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per half a processor (8 cores). The operating system is Ubuntu 14.04 (kernel version 3.16.0).

As in previous work [3, 4, 27], we measured the performance of the execution with flushes on a real system because we assume that an NVM will use a controller that will write data quickly into a local fast VM. We also assume that upon a crash, local batteries will allow saving the remaining local volatile data to the NVM. Thus, the actual flush cost is expected to be similar to the one we see on current platforms.

Since the queue is not a scalable data structure, executions with many threads are not relevant and we only measured 1-8 threads. Each execution lasted 5 seconds. All functions were implemented in C++ and compiled using the g++ compiler version 6.2 with the -O3 optimization flag. Memory management was handled with hazard pointers, and is described in Section 7. In our implementation, we followed the hazard pointer scheme provided in [22] and used the implementation of [23]. Following [19, 24], we evaluated the performance of the queue algorithms with a workload that lets several threads run enqueue-dequeue pairs concurrently. The queue is either initiated with 5 enqueued elements (for a small queue), or 1,000,000 enqueued elements (for a large queue). We depict the difference in the throughput of the MS queue and our three new algorithms across different numbers of threads. Each test was repeated 10 times and the average throughput

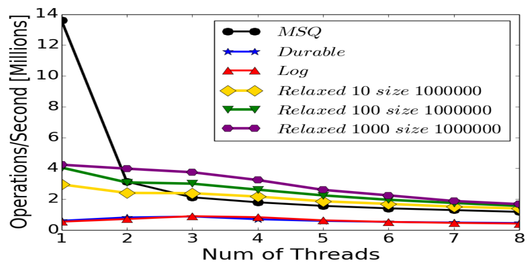

Figure 11. Throughput of the various queue implementations with no object reuse.

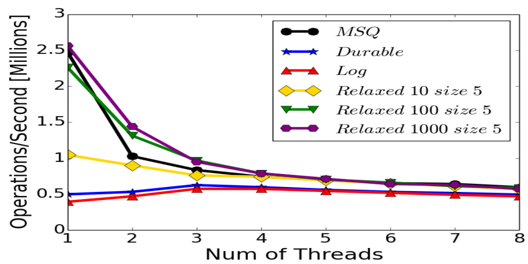

Figure 12. Throughput of the various queue implementations with memory management. Initial size of the queue is 5.

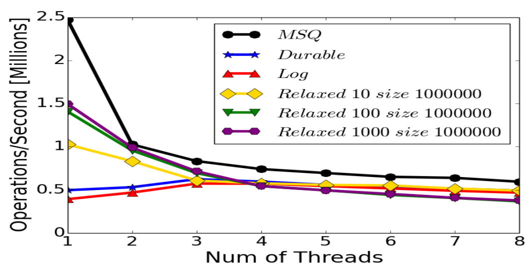

Figure 13. Throughput of the various queue implementations with memory management. Initial size of the queue is 1,000,000.

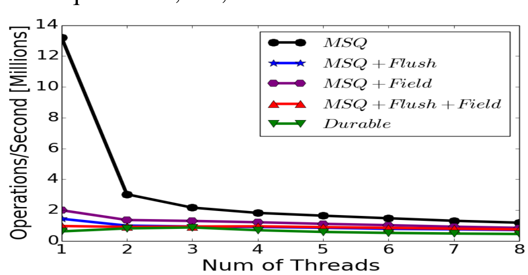

Figure 14. MSQ, only flushes added, flushes and additional fields, and the entire durable queue. Throughput reported in millions of operations per second.

is reported. The x-axis denotes the number of threads, and the y-axis stands for millions of operations per second. A high number is better, meaning that the measured scheme has higher throughput. With hazard pointers (designed in [22], and implemented in [23]) the memory management overhead is large and the results of Figure 12 and Figure 13 are less indicative of the bare queue actual performance. This is why we also provide the measurements without memory management in Figure 11. As expected, queues that provide weaker durability guarantees perform better in most cases, with the exception being when the queue is very large and the garbage collection costs dominate performance. We believe that the reason for this is that large queues employ many hazard pointers and this cost is similar to all queue variants. In contrast, small queues can avoid some flushes. When the sync() function is infrequently called, the new head may pass the old tail between snapshots, reducing the required number of flushes, and eliminating most of the hazard pointer uses. Surprisingly, the relaxed queue performs better than the MS queue without garbage collection. We believe this is due to an implicit back-off effect that the slower queue creates.

We ran the relaxed queue and let each thread execute the sync() function every $K*N$ operations, where K varies between, 10, 100, 1000 and 10000 and N is the number of the threads. We omitted the $K = 10000$ results because they are similar to the $K = 1000$ results. As each of the N threads executes a sync every $K*N$ operations, we get that on average a sync is executed in the system after each thread executes K operations.

Finally, we study how the flushes and the field that we added to the durable queue affect the performance. To this end, we measured an execution of the original MS queue and three intermediate versions between the original MS queue and the durable queue. The first intermediate version only adds flushes for enqueue nodes, as required by the durable algorithm. The second intermediate version maintains an extra field in the node in which the dequeuing thread writes its identity. This field is then properly flushed. The third intermediate version performs flushes on the queue nodes and also maintains an extra field in the node in which the dequeuing thread writes its identity. This extra field is then properly flushed according to the algorithm. Finally, the full durable queue is run, where, in addition to the above overhead, we also use an array for returned values and flush those when necessary. In order to measure these overheads with minimal interference, we measured those queue performances without memory reclamation, so only our additions would take effect. The results for the AMD platform are presented in Figure 14. We note that the most dominant operation is node flushing, which has a big impact on performance. Adding the extra field and the flushes for it is cheaper than flushing the enqueued nodes. The cost of the extra field is represented by throughput that is 6.6x and 1.4x lower when running, respectively, 1 and 8 threads. The cost of the flushes is represented by throughput that is 9.15x and 1.65x lower when running, respectively, 1 and 8 threads.

We also provide additional measurements on Intel similar to the ones reported for AMD. While the performance numbers are different, the trend is very similar. The platform is an 8-core **Intel** Xeon D-1540 2.6GHz with hyper-threading. This machine has 8GB RAM, an L1 cache of 32KB per core, an L2 cache of 256KB, and an L3 cache of 12MB. The operating system is Ubuntu 14.04 (kernel version 4.4.0).

The Intel version for Figure 11 is provided in Figure 15. The Intel version for Figure 12 is provided in Figure 16, and the Intel version for Figure 13 is provided in Figure 17. On the Intel platform with no memory management, the relaxed queue performs worse by 3.7x when running one thread and performing sync() every 10 operations, and by 2.6x when performing sync() every 1000 operations. When there are more than two threads running concurrently, it performs worse by 1.03x, when performing sync() every 10 operations, and better by 1.15x when performing sync() every 1000 operations. It usually slightly outperforms the original queue. The durable queue has 12.5x lower throughput while running one thread, and 3.14x lower throughput while running two threads. However, it improves as the number of threads increases, until it reaches a factor of 2.2x while running 8 threads. The log queue has 2.3x lower throughput when running 8 threads, almost the same as the durable, 3.9x lower throughput when running 2 threads, and 16.2x lower throughput when running one thread.

Finally, the Intel version for Figure 14 is provided in Figure 18. It seems that the node's flush is the most effective when 1-4 threads are running, but then the extra field is more dominant when 5-8 threads are running. The cost of the nodes flushes is represented by throughput that is 4.7x, 1.6x and 1.2x lower when running, respectively, 1, 4 and 8 threads. However, the cost of the extra field is represented by throughput that is 3.2x, 1.6x and 2x lower when running, respectively, 1, 4 and 8 threads.

9 Related Work

To the best of our knowledge, the presented queues are the first lock-free data structure designed for adapted execution with NVM. Several papers propose definitions for durability. In this paper we work with the definition of [17] but our algorithms and guidelines suit other definitions as well. In [27] the authors propose alternative definitions, some of which require hardware modifications. They also design a queue, but it is not lock-free. They use a lock (with additional flushes) to synchronize queue access.

Several prior works proposed transactional updates to persistent memory that guarantee *failure atomicity* – a collection of persistent data updates all occur or none do across failure boundaries [4, 7, 12, 18, 20, 33]. While these approaches work, they trade off performance for consistency in the face

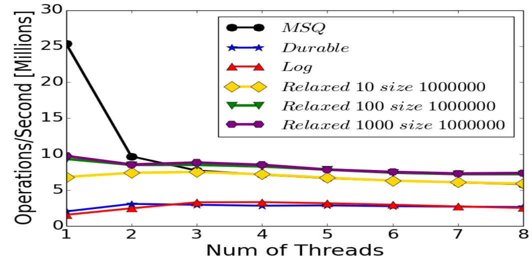

Figure 15. Throughput of the various queue implementations with no object reuse on the **Intel** platform.

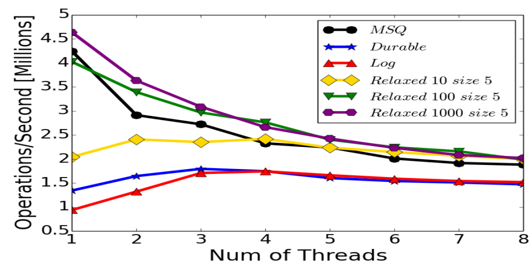

Figure 16. Throughput of the various queue implementations with memory management on the **Intel** platform. Initial size of the queue is 5.

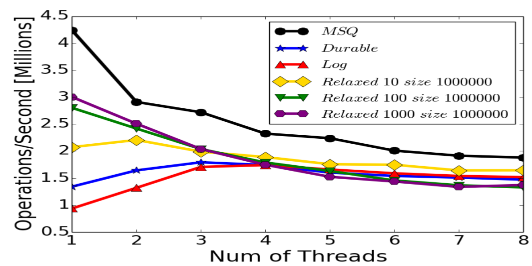

Figure 17. Throughput of the various queue implementations with memory management on the **Intel** platform. Initial size of the queue is 1,000,000.

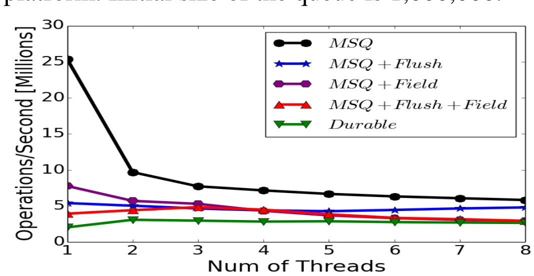

Figure 18. MSQ, only flushes added, flushes and additional fields, and the entire durable queue. Throughput reported in millions of operations per second on the **Intel** platform.

of failures – transaction runtimes incur significant bookkeeping overheads to consistently manage transaction metadata. An interesting alternative strategy to transactional updates is to build libraries of high performance persistent data structures [21] that are heavily optimized using ad hoc techniques informed by the data structure architecture and semantics. This is the focus of the current work.

Several other papers proposed using stable storage to maintain the state of the object. In [1] the authors propose solving consensus using stable storage by recording the state of the processes every round. Another paper [13] optimizes the logging procedure and provides a logarithmic lower bound for robust shared memory emulations.

Recently, [8] studied the construction of an efficient log adequate for non-volatile memory in the same settings as this work. This protocol can be extended to build an efficient single-threaded hash map. Additional related work was mentioned throughout the paper. 1

10 Correctness

A formal verification tool has been published by Derrick et al. [10]. Derrick et al. [10] uses this tool in order to formally prove that the Durable queue, which is presented in the paper, is durable linearizable [17].

11 Conclusion

In this paper we presented three designs for lock-free concurrent queues that can be used with non-volatile memory. These designs demonstrate avenues to deal with durable linearizability, buffered durable linearizability (with a sync operation), and detectable execution. As expected, full durable linearizability has a substantial performance cost. In contrast, buffered durable linearizability incurs a lower overhead as long as the sync operations are infrequently called.

References

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 2000. Failure Detection and Consensus in the Crash-recovery Model. *Distrib. Comput.* 13, 2 (April 2000), 99–125. https://doi.org/10.1007/s004460050070

[2] Marcos K Aguilera and Svend Frølund. 2003. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241* (2003), 25. http://www.hpl.hp.com/techreports/2003/HPL-2003-241.pdf

[3] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 677–694. https://doi.org/10.1145/2983990.2984019

[4] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[5] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *PVLDB* 8, 5 (2015), 497–508. http://www.vldb.org/pvldb/vol8/p497-chatzistergiou.pdf

[6] Ping Chi, Wang-Chien Lee, and Yuan Xie. 2014. Making B^+-tree Efficient in PCM-based Main Memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED '14)*. ACM, New York, NY, USA, 69–74. https://doi.org/10.1145/2627369.2627630

[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 *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[8] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient Logging in Non-volatile Memory by Exploiting Coherency Protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 67 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133891

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[10] John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2019. *Verifying Correctness of Persistent Concurrent Data Structures*. 179–195. https://doi.org/10.1007/978-3-030-30942-8_12

[11] Fatcache 2013. twitter/fatcache: Memcache on SSD. *https://github.com/twitter/fatcache*. (2013).

[12] Ellis Giles, Kshitij Doshi, and Peter J. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, Vol. 00. 1–14. https://doi.org/10.1109/MSST.2015.7208276

[13] Rachid Guerraoui and Ron R. Levy. 2004. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *in: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems, ICDCS*. 400–407.

[14] Maurice Herlihy. 1990. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming (PPOPP '90)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/99163.99185

[15] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[16] Intel-Architecture-Manual 2017. Intel Architectures Software Developer Manual. *https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf*. (2017).

[17] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects under a Full-System-Crash Failure Model. In *DISC 2016, Paris, France, 2016*.

[18] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[19] Edya Ladan-Mozes and Nir Shavit. 2008. An optimistic approach to lock-free FIFO queues. *Distributed Computing* 20, 5 (01 Feb 2008), 323–341. https://doi.org/10.1007/s00446-007-0050-0

[20] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred Persistence: Efficient Transactions in Persistent Memory. *Trans. Storage* 12, 1, Article 3 (Jan. 2016), 29 pages. https://doi.org/10.1145/2851504

[21] Managed-Data-Structures 2016. Hewlett Packard Labs: Data structures managed like never before on The Machine. *https://www.youtube.com/watch?v=3fN5_Qt9OCs*. (2016).

[22] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[23] Maged M. Michael. 2017. Hazard Pointers. *https://github.com/facebook/folly/tree/master/folly/experimental/hazptr*. (2017).

[24] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*. 267–275.

[25] Iulian Moraru, David G. Andersen, Michael Kaminsky, Nathan Binkert, Niraj Tolia, Reinhard Munz, and Parthasarathy Ranganathan. 2011. *Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores*. Technical Report CMU-PDL-11-114. Carnegie Mellon University.

[26] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[27] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2015. Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies. *IEEE Micro* 35, 3 (2015), 125–131. https://doi.org/10.1109/MM.2015.46

[28] RabbitMQ [n. d.]. RabbitMQ – Messaging that works *https://www.rabbitmq.com/*. ([n. d.]).

[29] Redis [n. d.]. Redis – in-memory data structure store, *http://redis.io/*. ([n. d.]).

[30] Swift [n. d.]. Swift Object Store. *https://swift.openstack.org/*. ([n. d.]).

[31] TuxedoMQ [n. d.]. Oracle Tuxedo Message Queue *http://www.oracle.com/us/products/middleware/cloud-app-foundation/tuxedo/message-queue/overview/index.html*. ([n. d.]).

[32] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. 5–5.

[33] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[34] WebSphereMQ [n. d.]. WebSphere MQ – IBM MQ. *www.ibm.com/software/products/en/ibm-mq*. ([n. d.]).

[35] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/2851141.2851168