Electronic Preprint for Journal of Information Processing Vol.29

Regular Paper

Durable Queue Implementations Built on a Formally Defined Strand Persistency Model

Jixin Han[1,a)] Keiji Kimura[1,b)]

Received: April 19, 2021, Accepted: August 6, 2021

Abstract: Emerging byte accessible non-volatile memory (NVM), or persistent memory (PM), technologies can promise durability like existing file systems even at an unexpected crash, as well as the competitive performance with DRAM. Similar to the memory consistency problems, appropriate order of memory access operations and cache eviction operations, or persistent operations, must be considered to guarantee both program recoverability and performance with the underlying persistency model. Several persistency models have been proposed in the literature. The strand persistency model, which potentially shows higher performance than the epoch persistency model, has more relaxed rules to exploit more parallelism. However, due to the lack of formal definition of the strand persistency model, legality and recoverability of strand persistency based programs against system crashes have been abandoned. To address this, we first propose an operational semantics of the strand persistency model to formalize the behavior of a program, memory propagation, and history generation under a concurrent environment. Then, we investigate the durability of library implementations for concurrent objects equipped with strand primitives, and propose a correctness criterion that the implementations should preserve, originated from buffered durable linearizability. Finally, as a case study, we discuss two concurrent queue implementations and show how the proposed semantics and criterion capture both the durability and linearizability of implementations.

Keywords: non-volatile memory, persistent memory, persistency model, buffered durable linearizability

1. Introduction

The emergence of nonvolatile, byte-addressable memory (NVM) technologies is significantly changing how the persistent data are handled nowadays. NVM, or persistent memory (PM), shows the characteristic in combining the feature of non-volatile storage in data recoverability after system crashes, and high performance in data transfer similar to ordinary DRAM. Various memory devices, such as Phase Change Memory (PCM) [15], [23], memristors [28], Intel/Micron 3D XPoint, etc, have been actively developed to realize PM above.

PM is potentially employed in future computer systems. It is placed on a memory bus and accessed by CPU cores directly by ordinary load and store instructions. Thus, anytime an unexpected crash happens, a program can recover the execution by inspecting the current status in the PM and roll-back or roll-forward to the correct stage of its execution.

However, a correct recovery after a crash requires that PM is maintained in a consistent state. That is, an inconsistent state may results in an incorrectly recovered program status. Maintaining PM in a consistent state is challenging, particularly with manipulating complex data structures in a multi-threaded environment.

One of the difficulties lies in the ordering constraint in making the data to a persistent state in PM, which is known as *memory persistency* (analogous to the traditional memory consistency). Generally, volatile memory modules (VM), such as caches, are placed between CPUs and PM. The actual processing order of persistent operations, each of which propagates data from a cache to PM, might be different from the issue order by CPU cores. This is because the current micro architectures try to change the memory access orders aggressively to hide slow low-level memory accesses. Another difficulty lies in guarantee the durability, which means data are required to reliably reach persistence. However, since the latency of PM is slow, the crash may happen before PM is updated, even after CPU cores have issued the operation and the cache has been modified. These two issues introduce counter-intuitive memory behavior, thus resulting in posing the difficulty to design the durable program and the recovery mechanism since it is difficult to determine which stage in the program execution became persistent before a crash occurred.

Both memory persistency rules and recovery mechanism should be carefully designed to guarantee ordering constraint and durability. An intuitive example is applying enqueue/dequeue operations to a FIFO queue concurrently. For instance, the queue is originally [1;2] where "1" is enqueued before "2". Assume that two threads are trying to enqueue "3" and "4" separately and concurrently. Under the memory consistency rules for each thread execution, one possible result is [1;2;3;4]. However, if the ordering of persistence for them is not processed properly, after a crash and its recovery, the queue could be [1;4] or [1;2;3;3].

Generally, such guarantee is supported by low-level ISA programming model. By correctly inserting persistent primitives into a program, the ordering constraint can be archived as a form of stalling the succeeding instructions until the previous instructions complete. The mechanism behind these persistent primitives is

[1] Waseda University, Shinjuku, Tokyo 169–8555, Japan
[a)] kalfazed@asagi.waseda.jp
[b)] keiji@waseda.jp

similar to synchronization primitives such as a fence instruction, which is used to protect memory consistency. In the past years, several ISA-level persistency models have been proposed to constraint the ordering of memory persistent operations. These models are different in how the data are persisted to PM. *Strict persistency* [22] is known as the most intuitive persistency model where the ordering in PM is identical to the ordering in the volatile memory. The strict model is straightforward but sacrifice the performance. *Epoch persistency* [3], [12], [20] is a more relaxed model where the ordering constraint between two instructions can be neglected if they have no dependencies in the aspect of persistency and they are in the same *epoch*. Epoch model is currently used in the Intel memory persistency. However, such a model still contains unnecessary dependencies, which degrade performance. For example, relaxing the ordering constraint of two instructions, which are located in different epochs, is impossible even if they have no dependency.

Compared with strict and epoch models, *strand persistency* [6], [22] is the most relaxed model that seeks more opportunities to parallelize persistent behaviors in both inter and intra threads. In the strand persistency model, a thread execution is divided into several logically independent segments, named *strands*. For each strand execution, all prior ordering constraints in PM are cleared so that each strand behaves as if a separate thread, which provides more opportunities in parallelization than the epoch persistency model. Gogte et al. [6] constructed hardware mechanisms to implement the strand persistency model and shows better performance than the other two models.

Although the development of computer architectures and programming models on PM has grown rapidly, especially in the aspect of pursuing higher performance, few works have studied the correctness of a durable program. The legality of a program under a relax persistency model must be carefully designed to meet the durability and recoverability. From the view of programmers, it is interesting to know weather libraries of highly optimized, high performance data structure can always keep the data stay safe in PM with the help of ad hoc techniques whenever a system crashes unexpectedly. This observation has led to the design of the criteria about the correctness of a library implementation for concurrent objects on the premise of crash and recovery. *Buffered durable linearizability* [11] proposed by Izraelevitz et al. is generally known as the extension of traditional linearizaiblility [8], which serves as the conventional notion of the correctness for concurrent objects. Buffered durable linearizaibility governs the safety of concurrent objects under the failure model. Such condition has been widely used to prove the durability of library implementations such as MS queue, Hashmap, linked-list, etc. under epoch persistency [4], [24], [26]. However, to the best of our knowledge, no studies discuss the legality of a program under the strand model using such a correctness condition. Especially, no existing work formalize the behavior of the strand persistency model yet. As a result, programs under it are lack confidence in recovering to the correct execution state as PM is not guaranteed to be consistent.

In this paper, we discuss the legality of durable library implementations of a queue under the strand persistency model from a formal perspective. We propose to look at a queue because it is a frequently used, highly-concurrent data structure. It is classical and universal, whereas is complicated enough to demonstrate the challenges raised by concurrent durable data structures, and simple enough to demonstrate solutions. Prior works [4], [11], [12], [22], [24], [25] of memory persistency also tend to use queues as their examples. To discuss the legality of a program formally, we formalize the strand persistency model under TSO memory consistency and describe its behavior in operational semantics. We designed an abstract memory model for strand persistency and create a set of operational semantics to describe the behavior of strand primitives and concurrency under the strand persistency model, which is informally proposed in prior works [6], [14]. We then proposed the legality of a durable program using the buffered durable linearizability and the invariant of program and memory status. Finally, as a case study, we show two durable queue libraries and discuss the legality.

In summary, we made the following contributions in this paper:

- We formally define the behavior of strand primitives in the operational semantics style, describing how the strand buffer, volatile memory, and PM are modified by executing these primitives.
- We use buffered durable linearizability to define the legal behavior of a program under the strand persistency model.
- We use two queue implementations under the strand persistency model as a case study to discuss the legality of durable library implementations using undo-logging under TSO and the strand persistency model.

This paper serves as the start point of the verification of programs under the strand persistency model. Formalized behavior of it can be used to develop reasoning techniques for verifying the correctness of persistent programs and libraries, and to develop program logics that would allow us to verify properties of a persistent program. However, the formal verification of a program written in the strand persistency model, and development of logics around memory persistency are excluded in the scope of this paper. As far as we know, we are the first to regulate the semantics of the strand persistency model and study the durability and ordering constraint of library implementations using the strand persistency model. We believe that strand persistency model is an efficient model that provides more opportunities to achieve better performance in the future.

The rest of this paper is organized as follows. Section 2 briefly introduces the motivation of the paper, including the importance, significance, and difficulties of holding PM in a consistent state, and why we need to focus on the legality of a program that use the strand persistency model. Section 3 introduces the operational semantics for the strand persistency model, including the abstract memory structure. Section 4 makes a brief introduction of traditional techniques to prove the durability and linearizability. Section 5 describes how we extend the traditional concepts of durability, linearizability and legality to fit the strand persistency model. Besides, we choose two strand library implementations to discuss the legality. Finally, Section 6 concludes this paper.

2. Background and Motivation

This section describes the background and motivation of this paper. We use an intuitive example to show the importance and difficulties in holding PM in a consistent state when a system crashes unexpectedly. Besides, we explain the reason why we focus on the strand persistency model by making a comparison with some other persistency models.

2.1 Durability and Ordering Constraint in PM

Maintaining PM in a consistent state is crucial in recovering a program into a proper stage of its execution when a system crashes. Otherwise, the recovered program may result in undefined or uncertain behaviors. Such consistency is guaranteed by ensuring the durability and ordering constraint in PM. Durability of PM makes sure the data are reliably persistent, whereas ordering constraint keeps the writes to PM are organized in a proper way.

However, guaranteeing the durability and ordering constraint on PM is difficult, especially in a concurrent environment. One of the reasons comes from the long latency taken by an PM update. Here is an example of a problem: Assume that two threads are executed concurrently in a system. One of the threads tries to update an object on PM, and a crash happens before the object on PM is actually updated. However, the other thread was acting as if PM had been updated right before the crash. After the system is recovered, PM state will be inconsistent from the view of these two threads.

Algorithm 1 is an intuitive example to show such a problem in applying the enqueue to the Michael-Scott queue (MSQ) [19] concurrently. This implementation has one *st* instruction in L_2 and three atomic memory operations *CAS* in L_9, L_{13}, and L_{17}. This example is safe and the queue is guaranteed to be always consistent in volatile memory, but such consistency might be lost in PM because writing back data from cache to PM takes time. First, from the view of intra-thread, since *CAS* in L_9 and L_{17} has data dependency with *st* in L_2, CPU can issues writeback of L_9 or L_{17} only after L_2 is issued. However, the data in L_2 might have not reached PM whereas L_9 or L_{17} have finished the writeback. This results in the fact that the pointer of queue points to an unallocated memory segment, which destroys the consistency of the data structure in PM. From the view of inter-thread, *CAS* cannot guarantee the atomicity of the queue in PM because the order that each *CAS* complete in PM might be different with the order that each *CAS* is issued. For instance, two threads T_1 and T_2 execute the enqueue for MSQ concurrently, the order that CPU issues *CAS* is $T_1.L_9; T_1.L_{17}; T_2.L_9; T_2.L_{17}$. However, the order that these instructions complete in PM might be $T_1.L_9; T_2.L_9; T_2.L_{17}; T_1.L_{17}$, which means the tail of queue is from the T_1 but the tail-> next is from T_2. If the system crashes, the program will apply recovery algorithm based on the current status of PM. However, the queue structure has been destroyed by inappropriate persistent ordering.

Another reason for difficulty is the order of the data persistent operations in PM. The current microarchitectures try to reorder instructions, including memory operations. Thus, the order of the data persistent operations might be different from that for the volatile memory. Since PM persistent operations have longer latency than the store operations for volatile memory, a more relaxed memory ordering constraint is required. Unlike the ordering constraint arising from memory consistency, which prescribe the visible order of loads and stores among threads, the ordering constraint in the data persistent operations guarantees the ordering of persists with respect to system failure atomicity. Thus, the persistent ordering provides more parallelism as long as the minimal set of happens-before relations in memory consistency are kept. Generally, the ordering constraint for memory persistency is realized in ISA-level programming by low-level primitives. For example, Intel [9] uses *sfence* instruction to enforce a strict ordering that any persist after the *sfence* cannot start until all persists before the *sfence* finish the execution.

Algorithm 1: Enqueue operation for undurable MSQ

```
   Data: value
1  node = new_node());
2  node -> value = value;
3  node -> next.ptr = NULL;
4  while true do
5      tail = Q -> Tail;
6      next = tail.ptr -> next;
7      if tail == Q -> Tail then
8          if next.ptr == NULL then
9              if CAS(&tail.ptr->next, next, <node, next.count+1> then
10                 break;
11             end
12         else
13             CAS(&Q->Tail, tail, <next.ptr, tail.count+1>);
14         end
15     end
16 end
17 CAS(&Q->Tail, tail, <node, tail.count+1>)
```

2.2 Low-level Persistency Models

Here, we use an example to review three low-level persistency models described by Pelley et al. [22]. They are different in the ordering constraint of persistent operations.

2.2.1 Strict Persistency

The strict persistency model is known as the most intuitive way to organize the ordering in PM. In this model, the ordering of persistent operations on PM is identical with that on the volatile memory. As similar as the sequential consistency that *store*s in each thread cannot be reordered from the perspective of other threads, all the persists should follow the same ordering.

In other word, the order that each *store* becomes durable in PM is completely same as the order that they become visible in the cache. This model sacrifices the performance since even unnecessary ordering constrains, or dependencies, among persistent operations for the recovery must be preserved.

In **Fig. 1**, (a) shows a high-level program with three assignments. In this program, L_2 has data dependency with L_3, and L_1 is completely independent from both L_2 and L_3. As a result, when we execute this program in parallel, we can write it as $L_1 \parallel (L_2; L_3)$. (b) shows a transformed low-level program with the strict persistency model. As we only discuss the memory persistency here, *store* and *load* are not included in (b), (c), and (d).

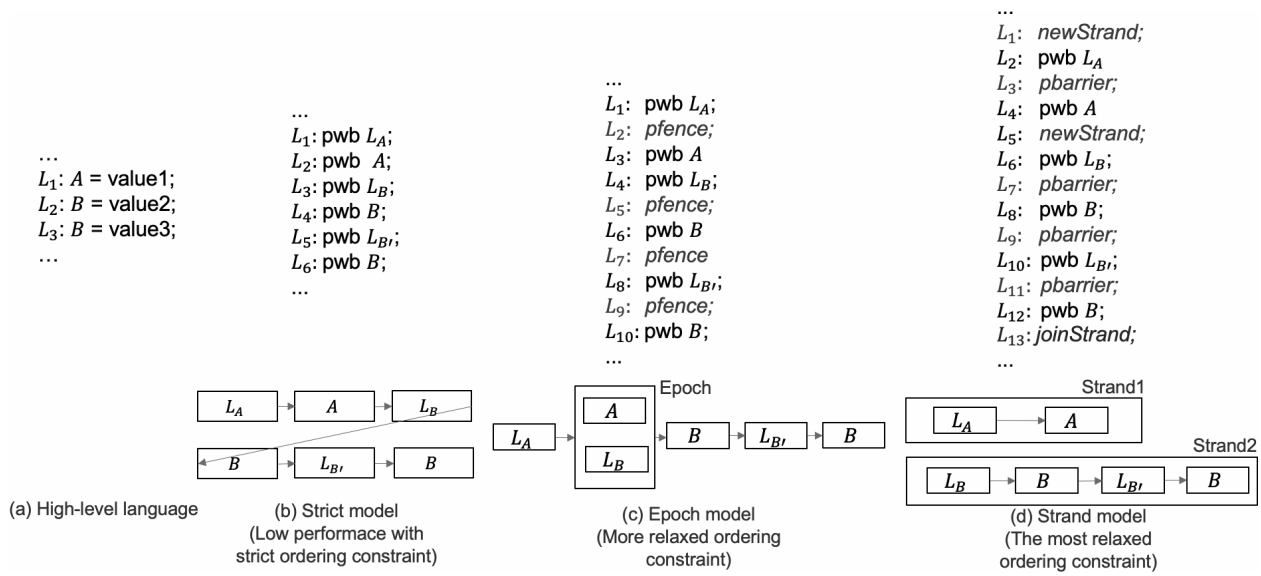

Fig. 1 Three types of memory persistency model.

In low-level program, we use "*pwb x*", which is short for "persist write back", to denote the write-back of x from the cache to PM. The behavior of *pwb* is similar with *CLWB* in Intel [9]. Lx is the log of variable x, which contains the old value of x before x is updated. The persist ordering of this program must keep that (1) Each Lx must be persisted before x, (2) The first persist of variable x must be persisted before the second one, and (3) If *pwb* of x and y are strictly ordered, the *pwb* of Lx and Ly should also keep such ordering. In the strict model, as each *pwb* cannot be issued until the previous *pwb* is completed, (1), (2), and (3) are guaranteed. However, it introduces unnecessary dependency over $L_1; L_3$ where they can be parallelized. This program can be written as,

$$L_1; L_2; L_3; L_4; L_5; L_6$$

2.2.2 Epoch Persistency

Compared with the strict persistency model, the epoch persistency model loosens the persist ordering constraint. Under epoch persistency, the execution of each thread is divided into several parts, each of which is called *epoch*, by a new low-level primitive named persist barrier, which is known as *pfence* in Intel. As *fence* is used to restrict the memory ordering in cache, persist barrier is used to restrict the memory ordering in PM. Any *pwb* after the persist barrier cannot be issued until all *pwb*s before the persist barrier are completed. All *pwb*s within a same epoch can be executed concurrently, whereas those in two epochs divided by a persist barrier must be executed in sequential. Besides, for the conflict access to the same address by multiple threads, the persist ordering respects the order of the corresponding stores. The epoch persistency model succeeded in exploiting more parallelism compared with the strict persistency model, and have been used in some current hardware design such as Intel x86 [25] and HOPS [20] recently. However, the parallelism among persistent operations from different epochs is neglected.

Figure 1 (c) shows a transformed low-level program with the epoch persistency model. In this program, since L_3 and L_4 can be parallelized, they are included in one epoch. The rest of this program keep the persist ordering in (b). The execution of this program is shown as,

$$L_1; (L_3 \parallel L_4); L_6; L_8; L_{10}$$

Though the persist ordering in (1) ∼ (3) are protected, it still introduces an unnecessary dependency between L_1 and L_4 because the *pwb* of Lx and Ly can be parallelized if x and y are different location.

2.2.3 Strand Persistency

The strand persistency model [6] inherits the characteristics of the epoch persistency morel whereas it can exploit more parallelism among persistent operations. This model newly introduces a new concept "strand", which is a kind of a fragment of a thread. The strand persistency model is supported by three new primitives, named *newStrand*, *joinStrand*, and *pbarrier*. *newStrand* is used to declare that the issue of all the *pwb*s following the *newStrand* is independent from any *pwb* before the *newStrand*. The program segment between two adjacent *newStrand* is one strand. In other word, the execution of persistent operations in different strands can be parallelized. Besides, *joinStrand* is used to synchronize the execution of strands. All the strands before the *joinStrand* must be completed before any *pwb* after the *joinStrand* can be issued. The persist ordering inside each strand is restricted by *pbarrier*, which is similar with the *pfence* in epoch model. As same as *pfence*, *pbarrier* is used to restrict the ordering constraint of *pwb*s, the *pwb* following *pbarrier* cannot be issued before the previous *pwb* is completed. However, *pbarrier* is a more relaxed primitive because the issue of *store* following *pbarrier* does not need to wait for the completion of previous *pwb*s, which increase more opportunity of parallelization.

Figure 1 (d) shows an example of strand model, which is originated from (c). The redundant dependency between L_4 and L_6 is removed. L_2 and L_4 are included in the first strand, L_6, L_8, L_{10} and L_{12} are included in the second strand. The execution of this program can be shown as,

$$(L_2; L_4) \parallel (L_6; L_8; L_{10}; L_{12})$$

2.3 High-level Persistency Models

ISA-level persistency models above are efficient but hard for programmers to design actual programs on them that satisfy the ordering constraints. It is burdensome and error-prone to directly manipulate persistency primitives, such as *pfence*, *newStrand*, etc. Besides, different architectures [9] provide architecture-specific instructions to guarantee the durability and ordering. One of optional ways is using libraries that build upon these ISA-level primitives [2], [16], [29] to ensure the failure atomicity [7], [13], [17] in the granularity of a transaction [14]. However, programmers still must carefully use them and easy to violate the required ordering constraint.

Recently, several works about the persistency models for language-level, such as C/C++, have been studied. The persistency models in these works are based on synchronization primitives and extend the semantics to meet failure-atomicity. For example, ATLAS [1] handles the failure atomicity of a critical section bounded by lock/unlock using undo-logging to make sure the recoverability of failure-atomic section. Other alternative approaches of using logging mechanism also have been proposed, such as redo-logging [21] or justdo-logging [10]. Besides, PMTest [18] provides assertion-like high-level checkers to guarantee the durability and persist ordering. These high-level implementation code are then mapped into ISA-level primitives for hardware structure to correctly order persistent operations, such as CLWB and SFENCE instructions in the Intel x86 ISA [9]. However, the mapped ISA-level program has stricter constraints that are over the programmers' expectation. Gogte et al. [6]. tried to relax such constraints by mapping into the strand persistency model and enable greater persist concurrency than the epoch persistency model.

3. Operational Semantics

We designed an operational semantics of the strand persistency model on top of TSO memory consistency [27]. The semantics design is inspired by the semantics of the epoch persistency model published by Raad et al. [24]. We picked theirs as the basement of our semantics design for the following reasons: (i) The strand persistency model reuses some of the concept of the epoch persistency model. For example, both of two models have the *persistent buffer* to stall pending persists. Since the strand persistency model acts as a more relaxed model, the semantics around the manipulation of persistent buffer should be designed to support more concurrency. (ii) The design of such a semantics is easy to be linked into the correctness condition for recoverability of a program, which refers to *"Buffered durable linearizability"*. By tracking the transition of a program and its memory status, we can easily determine the moment that the program takes effect on PM.

The formalized behavior of strand primitives in the operational semantics strictly respect the informal description of the strand persistency model proposed by Gogte et al. [6].

3.1 The Syntax of a Program

In this paper, a target program can use strand primitives to control the memory persistency in high-level. These primitives include *pbarrier*, *newStrand*, and *joinStrand*. Though they are designed for ISA-level program, programmers can manually insert them in their programs. The *pwb* instruction processes a write-back operation for a cache line, associated with the specified address as an argument, to PM. The *psync* instruction makes sure all the stores prior the *psync* should complete the write-back from cache to PM [9]. However, not all the stores need to be written back since some of them are not necessary in the aspect of recovery. By adding the argument to the *pwb* instruction, it makes possible to remove redundant writeback and improve performance. For other general instructions, such as if-else, skip, assignment, etc. we reuse the definition of them from Ref. [24].

3.2 Abstract Memory Structure

The strand model can be realized on top of the abstract memory structure that supports the behavior of a program, which has high potential of the concurrency of memory persistency. **Figure 2** shows the overview of the structure. Each thread (τ) has a Local counter (CT), Store queue (SQ), and Strand buffer unit (SBU). CT is used to count the memory operations issued by each thread, and it is used to establish the partial order among memory operations by providing time (T) to each operation. SQ is designed as a queue to keep the pending stores issued by a thread in the FIFO style. SBU is a list of Strand buffer (SB). Each SB has a Persist buffer (PB) and Persisting table (PT). PB is used to keep the *pwb*s issued by a thread in the FIFO style. As writing back to PM takes time to get response, PT is designed as a table to keep track of the running *pwb*s. As long as a *pwb* is issued and successfully persisted in PM, it can be removed from PT. PT is located alongside with the corresponding PB.

Cache (C) and Persistent memory (PM) are shared among threads. Both of them are mapping from memory location to the value. When a value reaches the cache, it becomes "visible" from all threads. Similarly, when a value in the cache is written back to PM, it becomes "durable", which means it can remain in PM even after a system crash. Different from the previous work discussing a semantics for the epoch persistency model, which introduces a "persistent buffer" to provide visibility [24], the cache has a responsibility for it in our abstract architecture. This is because the strand persistency model provides parallelism among strands and it is difficult to represent it by a single persistent buffer. Thus, we introduce the strand buffer units and the cache separately to manage it. The syntax is defined as below:

$$
\begin{aligned}
CT &::= threadID \xrightarrow{fin} T \\
SQ &::= threadID \xrightarrow{fin} OP \\
SBU &::= threadID \xrightarrow{fin} SB \\
C &::= loc \xrightarrow{fin} value \\
PM &::= loc \xrightarrow{fin} value \\
SB &::= Seq\langle PB, PT\rangle \\
PB &::= Seq\langle OP\rangle \\
PT &::= loc \xrightarrow{fin} value \\
OP &::= \langle st(loc, x) | pwb(loc) | pbarrier, T\rangle
\end{aligned}
$$

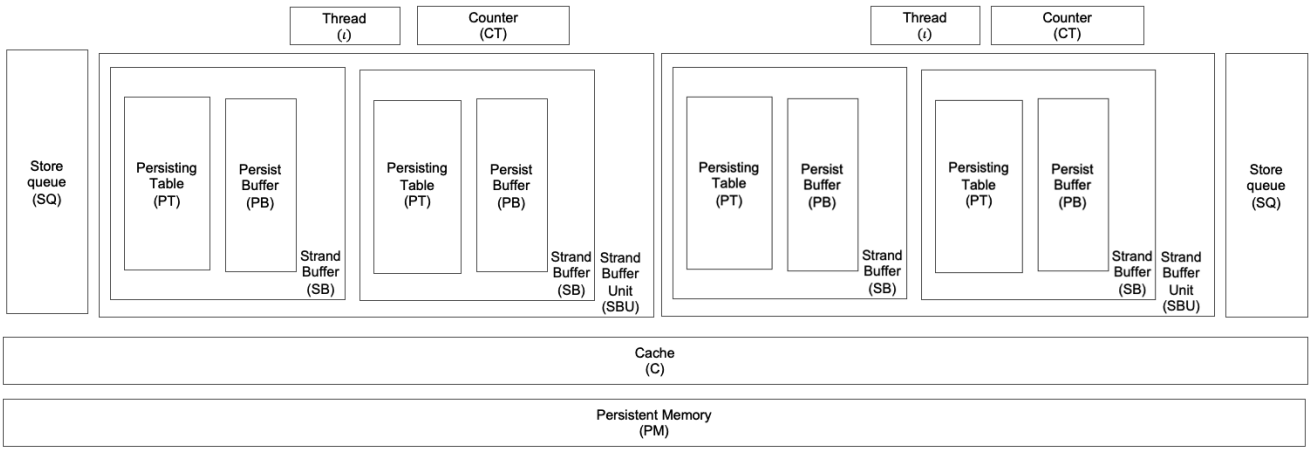

Fig. 2 The abstract memory structure of strand model.

The state transition of memory model $M \xrightarrow{\tau:l} M'$ follows the label l issued by thread τ. M stands for "Memory", which is a tuple of CT, SQ, SBU, C, and PM.

$M ::= \langle CT, SQ, SBU, C, PM \rangle$

Generally, l can be divided into three categories: (1) *memory manipulation operations*, which include $st(loc, x)$, $pwb(loc)$, and $CAS(loc, s, t)$, to update the memory structure deterministically. (2) *memory ordering operations*, which include *fence*, *pbarrier*, *newStrand*, and *joinStrand*, to keep the memory ordering constraint in both volatile memory and PM. (3) *silence operation*, which is denoted as ϵ, to show the non-determinisic update in the memory structure. The silence operations are activated when the system can perform the memory propagation. The detail will be introduced in Section 3.3. The memory model supports the concurrency with the help of silence operation. When multiple memory states share the same condition to perform a silence operation, they can be updated in a random order. For example, the updates of two difference locations in PM can be performed in any order if they have no dependency each other.

3.3 Semantics of Program and Memory Transition

A memory transition $M \xrightarrow{\tau:l} M'$ is a short for $CT, SQ, SBU, C, PM \xrightarrow{\tau:l} CT', SQ', SBU', C', PM'$. We designed a set of rules in the semantics ($R.1 \sim R.11$) to describe how the memory status is updated deterministically or non-deterministically. To simplify the semantics in this paper, we only shows memory units that have related to each semantics definition, others are omitted.

store (*R*.1): When a thread issues an $st(loc, x)$, that means the value of location *loc* in the cache will be updated into a new value x. As the memory model is built on top of TSO, it is possible to stall the pending *st*s in each store queue before they reach the cache. As a result, after a thread issues an *st*, it is appended at the end of the local store queue with the current local counter, and increment the local counter. A pending *st* in the store buffer will be propagated to the cache follows the semantics *R*.7.

CAS (*R*.2): Compare-and-Swap(CAS) is an atomic memory operation. When a thread issues a $CAS(loc, s, t)$, it will first load the value from the location *loc* in the shared memory unit (firstly check the cache, and then load from PM if no associated data in the cache). If the value is the same as s, the value of location *loc* in the cache will be updated into t. Before a *CAS* is issued, all the pending *st* issued before the *CAS* become visible, or update the cache. Thus, the store queue of the thread becomes empty. As long as a *CAS* is issued, it becomes visible instantly in the cache, and the local counter is increased.

pwb (*R*.3): When a thread issues a $pwb(loc)$, it is put into the latest strand buffer in the local strand buffer unit of this thread. As a strand buffer is a pair of a persist buffer and a persisting table, the *pwb* will be first appended at the end of the persist buffer with the current local counter, then waiting for the moment when the *pwb* can be propagated to PM. As well as an *st* and a *CAS*, the counter will be incremented after a *pwb* is issued.

pbarrier (*R*.4): The semantics around *pbarrier* is the same as *pwb*. Both of a *pbarrier* and a *pwb* are propagated to a persist buffer at the beginning. The difference is that the propagation path of a *pwb* is $PB \rightarrow PT \rightarrow PM$, whereas a *pbarrier* is only used to stall the propagation of a *pwb* in *PB*. The detail is shown in *R*.7 and *R*.10.

fence (*R*.5): The *fence* instruction is used to make sure all the *st*s in a store queue become visible to other threads before the thread issues *fence*. As a result, no more instructions can be appended to the store queue until the current store queue becomes drained.

newStrand (*R*.6): When a thread issues a *newStrand*, it means all the following memory operations related to the memory persistency have no dependency with previous strands. A new empty strand buffer will be added at the end of the strand buffer unit of this thread. Any *pwb* or *pbarrier* after the *newStrand* will be put into this new strand buffer.

joinStrand (*R*.11): The *joinStrand* is similar to a synchronization among strands. All the propagation must be finished before a thread can issue a *joinStrand*. That means all the *st*s' results become visible in the cache and all the *pwb*s' results become durable in PM. After issuing the *joinStrand*, the thread will reset its local counter to 0.

propagation (*R*.7 – *R*.10): Different from all previous semantics, the propagation timings of an *st* and a *pwb* are nondeterministic. For example, as long as an *st* exists in the store queue and it has no dependency, which prevents it from becom-

ing visible, the *st* can be propagated to the cache at any time. **stPropC** (*R*.7) restricts the propagation of an *st*. Firstly, an *st* becomes visible following the FIFO order that they are kept in a store queue. Secondly, if any *pwb* is trying to write back the value to PM from the same *loc* with an *st* issued after the *pwb*, and a *pbarrier* has been issued after the *pwb* and before the *st*, then the propagation of the *st* cannot be started until the *pwb* starts its propagation. **pwbPropPT** (*R*.8), **pwbPropPM** (*R*.9), and **pwbStall** (*R*.10) restrict the propagation of a *pwb*. A *pwb*(*loc*) becomes ready to propagate if all *st*s that try to write the same *loc* to the cache before the *pwb*(*loc*) are visible (*R*.8). If so, the *pwb* will be first propagated to the persisting table and become pending, then be propagated to PM from the persisting table (*R*.9). Note that propagating of a *pwb* is different from that of an *st*. An *st* can become instantly visible as long as it is propagated from the store queue. However, as PM takes time to get updated, a *pwb* cannot be durable as soon as it is propagated. This is also the reason why the *pwb* propagation is divided into *R*.8 and *R*.9. Another restriction to propagate a *pwb* is a *pbarrier*. Before propagating any *pwb* following a *pbarrier* in the persist buffer, all the *pwb*s in the persisting table become durable (*R*.10).

$$\frac{l = st(loc, x), \quad CT(\tau) = t}{CT, SQ \xrightarrow{\tau:l} CT[\tau \mapsto t+1]), SQ[\tau \mapsto SQ(\tau).\langle l, t\rangle]} \quad (R.1(st))$$

$$\frac{l = CAS(loc, s, t), \quad CT(\tau) = t, \quad SQ(\tau) = \emptyset, \quad ld(PM, C, \emptyset, loc) = s}{CT, C \xrightarrow{\tau:l} CT[\tau \mapsto t+1]), C[loc \mapsto t]} \quad (R.2(CAS))$$

$$\frac{l = pwb(loc), \quad CT(\tau) = t, \quad SBU(\tau) = SBU(\tau)'.sb, \quad sb = \langle pb, pt\rangle, \quad sb' = \langle pb.\langle l, t\rangle, pt\rangle}{CT, SBU \xrightarrow{\tau:l} CT[\tau \mapsto t+1]), SBU[\tau \mapsto SBU(\tau)'.sb']} \quad (R.3(pwb))$$

$$\frac{l = pbarrier, \quad CT(\tau) = t, \quad SBU(\tau) = SBU(\tau)'.sb, \quad sb = \langle pb, pt\rangle, \quad sb' = \langle pb.\langle l, t\rangle, pt\rangle}{CT, SBU \xrightarrow{\tau:l} CT[\tau \mapsto t+1]), SBU[\tau \mapsto SBU(\tau)'.sb']} \quad (R.4(pbarrier))$$

$$\frac{l = fence, \quad SQ(\tau) = \emptyset,}{SQ, C \xrightarrow{\tau:l} SQ, C} \quad (R.5(fence))$$

$$\frac{l = newStrand, \quad SBU(\tau)' = SBU(\tau).sb_0, \quad sb_0 = \emptyset}{SBU \xrightarrow{\tau:l} SBU[\tau \mapsto SBU(\tau)']} \quad (R.6(newStrand))$$

$$\frac{\begin{array}{c} SQ(\tau) = \langle op, t\rangle, SQ', \quad op = st(loc, x), \\ sb \in SBU(\tau), \quad sb = (pb, pt), \quad pb = pb_1.pb_2, \\ \nexists t_1, t_2.\langle pwb(loc), t_1\rangle \in pb_1, \langle pbarrier, t_2\rangle \in pb_2, t_1 < t_2 < t. \end{array}}{SQ, C \xrightarrow{\tau:\epsilon} SQ', C(loc \mapsto x)} \quad (R.7(stPropC))$$

$$\frac{\begin{array}{c} sb \in SBU(\tau), \quad sb = (pb, pt), \\ pb = \langle pwb(loc), t\rangle.pb', \quad C(loc) = x, \\ \nexists t'.\langle st(loc, x), t'\rangle \in SQ(\tau), \quad t' < t \end{array}}{SQ, C, SBU \xrightarrow{\tau:\epsilon} SQ, C, SBU[\tau \mapsto (sb \mapsto (pb', pt(loc \mapsto x)))]} \quad (R.8(pwbPropPT))$$

$$\frac{sb \in SBU(\tau), \quad sb = (pb, pt), \quad pt(loc) = x}{SBU, PM \xrightarrow{\tau:\epsilon} SBU[\tau \mapsto (sb \mapsto (pb, pt(loc \mapsto \emptyset)))], PM[loc \mapsto x]} \quad (R.9(pwbPropPM))$$

$$\frac{sb \in SBU(\tau), \quad sb = (pb, pt), \quad pb = \langle pbarrier, t\rangle, pb', \quad \forall loc.pt(loc) = \emptyset}{SBU \xrightarrow{\tau:\epsilon} SBU[\tau \mapsto (sb \mapsto (pb', pt)]} \quad (R.10(pwbStall))$$

$$\frac{l = joinStrand, \quad CT(\tau) = t, \quad SQ(\tau) = \emptyset, \quad \forall sb \in SBU(\tau), sb = \emptyset}{CT, SQ, SBU, C, PM \xrightarrow{\tau:l} CT[\tau \mapsto 0]), SQ, SBU, C, PM} \quad (R.11(joinStrand))$$

3.4 Persist Ordering in Strand Model

In the strand persistency model proposed by Gogte [6], the behavior of each strand primitive restricts the order of the cache and PM update operations.Such ordering constraints are shown as VMO (Volatile memory order) and PMO (Persist memory order). The behavior of *pbarrier*, *newStrand*, and *joinStrand* proposed in this paper strictly follows the definition in Gogte's work. They use following notations to describe the ordering constaint: M_x^i means a load or a store operation to the location x of PM on thread i; S_x^i means a store operation to the location x of persistent memory on thread i; PB^i means a persist barrier issued by thread i; NS^i means a *newStrand* issued by thread i; JS^i means a *joinStrand* issued by thread i; $M_x^i \leq_v M_y^i$ means that M_x^i is ordered before M_y^i in VMO; $M_x^i \leq_p M_y^i$ means that M_x^i is ordered before M_y^i in PMO;

First, Gogte described that *newStrand* and persist barrier restrict the VMO and the PMO of a program in the following way:

$$(M_x^i \leq_v PB^i \leq_v M_y^i) \wedge (\nexists NS^i : M_x^i \leq_v NS^i \leq_v M_y^i)) \rightarrow M_x^i \leq_p M_y^i$$

Such an ordering constraint can be performed by *R*.4, *R*.6, and non-deterministic memory propagation from *R*.7 ∼ *R*.10. From the definition in *R*.6 and *R*.8, as long as a program encounters a *newStrand*, a new strand buffer will be created and all the following *pwb*s are propagated in this strand buffer. Otherwise, *pwb*s will be propagated to the current strand buffer. From *R*.3 and *R*.4, a *pwb* and a *pbarrier* are issued to a persist buffer in the same way. Since a persist buffer is a FIFO buffer, both a *pwb* and a *pbarrier* are ordered in the same way. This is how the VMO part of the rule above is displayed in our model. However, because of the restriction of *pbarrier* in *R*.10, a *pwb* following a *pbarrier* in a persist buffer have to stall the propagation to PM until all *pwb*s before the *pbarrier* have finished the propagation to PM. This is how PMO part of the rule above is displayed in our model. As a result, the behavior of *pwb*, *pbarrier*, and *newStrand* respects the

behavior defined in Ref. [6].

Next, the VMO and the PMO introduces by *joinStrand* is shown as the following way:

$$(M_x^i \leq_v JS^i \leq_v M_y^i) \rightarrow M_x^i \leq_p M_y^i$$

Such ordering constraint can also be shown in our model. From *R*.11, when a program encounters a *joinStrand*, the program will stall the issue of new memory and persist instructions to both a store queue and a strand buffer unit until all the running memory propagations are finished. This is the VMO part of the rule above. As long as all running memory propagation are done, which means there is no pending *st* in a store queue, and no pending *pwb* in a strand buffer, all *pwb*s before the *joinStrand* are durable. All the *pwb*s following the *joinStrand* will be persisted after these *pwb*s. This is how the PMO part of the rule above is displayed. In the consequences, we say the *joinStrand* in this paper respects the rule defined in Ref. [6].

3.5 History Generation

To trace a program execution, we borrow the notion of *event* and *history* from Refs. [8], [11], which are originally used to show the correctness condition for a concurrent object. We use these two concepts to record the execution status.

$$\begin{aligned} H &::= TE \\ TE &::= (threadID \xrightarrow{fin} E) | c \\ E &::= inv(funcID) | res(funcID) \\ &\quad | st(loc, x) | CAS(loc, s, t) | pwb(loc) \end{aligned}$$

History (H) is defined as a list of thread events (TE), TE is either a crash event (c) or a mapping from a thread id to the event (E). Two types of event are defined: (1) Invocation and response. $inv(funcID)$ is an invocation event for a library call. When a thread issues $call(\mu)$ to execute a library call μ, $inv(\mu)$ is appended to the end of H. In contrast, $res(funcID)$ stands for a response event of a library call, it is appended to the end of H when a thread issues *return*. (2) Memory events. $st(loc, x)$ and $CAS(loc, s, t)$ are added to H as long as they become visible following the semantics *R*.2 and *R*.7. $pwb(loc)$ is added to H when it becomes durable, which means the *pwb* is completed following the semantics *R*.8 and *R*.9.

The transition rule for the history generation is defined as $\langle P, M, H\rangle \xrightarrow{\tau:l} \langle P', M', H'\rangle$. The update of H strongly depends on the semantics of the memory transition. We say a library call μ is "pending" if $\tau.inv(\mu) \in H$ but $\tau.res(\mu) \notin H$. A library call μ is "complete" if both $\tau.inv(\mu)$ and $\tau.res(\mu)$ are included in H.

4. Correctness Condition of Undurable Implementation [8], [11]

In a concurrent environment, linearizability is known as the correctness condition for the operations performed to a concurrent object, such as queue, hash table, etc. It is used to prove the legality of a library call implementation for a concurrent object. Buffered durable linearizability is an extension of such criteria to support both the linearizability and durability in a case of a system crash. An implementation that meets buffered durable linearizability means a program can recover to the correct state when a crash happened at anytime. In this section, we first make a brief introduction of linearizability [8] and the traditional verification techniques to prove an undurable implementation. We will describe the buffered durable linearizability and discuss how we extend these concepts to define the correctness of durable implementations in Section 5.

4.1 Linearizability [8], [11]

In linearizability, an operation in the granularity of a library operation performed by each thread is divided into an invocation and a response. The operation takes effect between the corresponding response and its invocation. Then, the behavior of a program can be simulated as a list of invocations and responses labeled with each thread id and properties of operations. Such a list is called *History*. The verification of the linearizability of a program can be done by verifying if the value of the shared object after the history is legal or not.

However, different from a sequential object execution, operations are allowed to overlap with each other and the ordering among operations can be non-deterministic in a multi-thread environment. This leads to the difficulty in verifying the correctness of the object value because the history can be interleaved. To identify the correctness of the history under such a non-deterministic situation, it should be convergence with the help of linearizability. The history H is sequential if each invocation, except possibly the last, is immediately followed by a matching response. Each response is immediately followed by a matching invocation. Here, we borrow some notion from Ref. [11] to define the linearizability. We say an operation is "pending" if the response of the operation is not included in H. An operation is "complete" if H contains a pair-wise invocation and response of the operation. $compl(H)$ denotes the set of histories that can be generated from H by appending completion responses, and $trunc(H)$ denotes the set of histories that can be generated from H by removing pending invocations. H induces an irreflexive partial order $<_H$ on operations to capture the "real-time" partial order between operations.

A history H is then can be recognized as linearizable with the following definition:

Definition 1 (Linearizability). *History* (H) *can be linearizable if H can be transformed into another history* (H') *with following rules: (1)* H' *is* $trunc(compl(H))$ *and is equivalent with some sequential history* H_s*. (2)* H_s *has the same happens-before with H.*

4.2 Correctness of Undurable Implementation

An implementation can be recognized as a set of histories, each of which consists of events of two objects, a representation object *REP* and an abstract object *ABS*. The operation events performed on *REP* and *ABS* are called rep operations and abs operations, respectively. Informally, rep operations are the instructions that build the implementation (e.g., store, CAS, etc), and abs operations can be treated as the function declarations of the implementation (e.g., enqueue, dequeue, etc), and the invocation and

response of these function declarations. Thus, it is clear to know that an abstract operation is implemented by the sequence of rep operations that occur within it.

We say a history H is *sequential* if: (a) The first event of H is an invocation. (b) Each invocation, except possibly the last, is immediately followed by a matching response, each response is immediately followed by a matching invocation. Besides, a history H is well-formedness if each sub-history $H|T$ is sequential, where $H|T$ means the sub-history of H including only the operations of thread T. In a history, events can be interleaved under the following restrictions: (1) the sub histories $H|REP$ and $H|ABS$ satisfy the usual well-formedness conditions, and (2) for each thread T, each rep operation in $H|T$ lies within an abstract operation in $H|T$.

When an implementation is executed in a multi-thread environment, concurrent execution to the same memory location will be serialized in the instruction level at real-time. Thus, $H|REP$ is linearizable by default. However, the linearizability of $H|ABS$ totally depends on if the $H|REP$ can result in a legal value where the legality respects the specification of abstract operation (ABS). Besides, such a value must also be one of the possible results of $H|ABS$.

We can define the correctness of undurable implementation naively and informally in the following way:

Definition 2 (Correctness of undurable implementation). *An undurable implementation is correct with respect to the specification of ABS if for every history H in the implementation, H|ABS is linearizable.*

4.3 Invariant and Abstract Function

We make a brief introduction of the traditional verification technique for an undurable implementation here.

The verification of a concurrent implementation requires an appropriate definition of an invariant and an abstract function. An invariant is a boolean value, which shows immutable properties that the target data structure must hold during the execution. An abstract function provides a mapping rule to show the data structure that satisfies an invariant.

A subset of REP values that are legal representations is characterized by a predicate named the "rep invariant", $I : REP \rightarrow BOOL$. The meaning of a legal representation is given by an abstract function $A : REP \rightarrow ABS$, defined for representation values that satisfy the invariant.

In a concurrent environment, as abstract operations are allowed to overlap with each other, an implementation must be prepared to encounter a rep value reflecting the incomplete effects of concurrent operations. As a result, an abstraction function must be defined continually, which means it is not defined between abstract operations but between rep operations that implement the abstract operations. Therefore, it is common that a rep invariant must be preserved by each rep operation in the sequence implementing each abstract operation. On the behalf of an abstract function, the non-deterministic behavior of a concurrent execution must be reflected to show an acceptable behavior of a concurrent program. For example, when two threads concurrently enqueue x and y in a non-deterministic way, both $[x; y]$ and $[y; x]$ are acceptable results of the program. As a consequence, an abstract function should be defined to map a rep value to a set of abstract values. This set represents all the possible sets of linearization permitted by the current value of the rep.

To verify the correctness of an implementation, it is helpful to transform assertions about interleaved histories into assertions about sets of sequential histories, which can be also known as sets of values. They are convenient to use and verify by using familiar axiomatic methods such as Hoare logic and Separation logic. For a given history H, the value of an object at the end of a linearization of H is called linearized value $Lin(H)$. As discussed before in this section, a history can have more than one linearization, so $Lin(H)$ is used to denote the set of all linearized values of H.

From the view of $H|ABS$, if an operation is invoked before the previous operation gets response, which means the execution of two operations are overlapping, the $Lin(H|ABS)$ contains several possibilities. However, as $H|REP$ contains instructions that implement the operation, each $Lin(H|REP)$ only contains one possible value of concurrent object. If an implementation is linearizable, it requires that $Lin(H|REP)$ is one of the possible result of $Lin(H|ABS)$. Besides, it also requires that the rep invariant I are always preserved in $H|REP$. The assertion of implementation needs to show these conditions are always met. It can be defined as following:

$$\exists r.r \in Lin(H|REP) \wedge r \in Lin(H|ABS) \wedge I$$

For example, assume that a queue q with one element $[t]$, and thread 1 performs $Enq(q, x)$ and thread 2 performs $Deq()$ concurrently. The execution of two operations are overlapping. In this case, $Lin(H|ABS)$ is $\{[t], [x]\}$. If the implementation of Enq and Deq are correct, $H|REP$ will exist that can generate either $[t]$ or $[x]$. However, if the implementation is incorrect, $H|REP$ may generate $[t; x]$ or $[]$, which disobey the assertion above. Then the implementation is proved to be invalid.

5. Durable Queue Implementations

In this section, we study the legality of library implementations for a concurrent object using the strand persistency model. Note that the correctness for the durable implementation should be carefully defined in the aspect of both durability and recoverability. Generally, three aspects are required in the criteria: (1) a crash may happen anytime, and a system should be correctly recovered; (2) a recovery should meet failure atomicity where none or all the updates in failure atomic region can be recognized; (3) The persist ordering after a crash strictly respects the state before the crash. In Section 4, we described the basic concept of the buffered durable linearizabilty (BDL). In this paper, we combine the knowledge of the axiomatic verification for linearizability with the idea of BDL and define a criteria for durable implementations. Such criteria can be recognized as an extension of the traditional BDL. It regulates the execution of a library implementation by showing the invariant of concurrent object and persistent memory status. Such invariant must be satisfied whenever a system crashes. Besides, the value of a concurrent object in PM must be legal. In the later part of this section, we use two case studies of queue implementations to show how we use this

criteria and semantics of the strand persistency model to discuss the legality of durable implementations.

5.1 Premise of Implementation and System Crash

Before stepping into the discussion of the legality of a durable implementation, we first describe the premise of the implementation of libraries and unexpected system crash to show the scope of this paper.

Tolerated failure and hardware premise First, we assume the crash is *fail-stop*. When a system crashes, as long as data are persisted to PM, these data are kept still and no process can destroy them anymore. The transient data, such as the data in thread stacks or caches, will be lost if they have not reached PM before a crash. We also consider a crash is *full-crash*, which means we assume all threads exit the execution when a system crashes. We exclude the case that partial threads are not affected by a system crash and continue execution. Next, we say all addresses in PM are required to be mapped into the same virtual address. Besides, we also assume the hardware architecture design follows the abstract memory model described in Section 3 and all CPUs are able to access the shared memory modules even after the crash. Lastly, each data employs per cache line to store so that when a CPU decides to flush a data, it flushes the whole cache line.

Strand model As described in Section 2, the strand persistency model has more relaxed persist dependencies within and across threads compared with the epoch persistency model. In the epoch persistency model, only consecutive persists within a thread can be labelled as concurrent. Besides, persists from different threads are only concurrent if their epochs race or if they are not synchronized. Many persists within and across threads can still be concurrently executed even if they do not fit these patterns. With the help of the strand primitives, more concurrency can be discovered in both intra-thread and inter-thread levels.

Undo-logging We apply the undo-logging mechanism designed in ATLAS [1] for recovery. Undo logging ensures failure atomicity by recording the undo-log of data before an in-place update in a failure-atomic region. The undo-log is reset after the corresponding in-place update in PM. If a system crashes before the new data are updated in PM, the recovery process will use undo logs to roll-back partial PM updates. If the system crashes after the updates in PM, the program will roll-forward from the latest updates in the undo logs. To ensure correct recovery, it requires the partial persist ordering between the undo log and the corresponding in-place update in PM. That is, the log of a location must be persisted before the in-place update of that location in PM. The undo logs and in-place updates for different memory locations can be persisted concurrently if their corresponding stores are concurrent. We use the strand persistency model to seek for the parallelism aggressively for undo-logging mechanism.

TSO We choose "total-store-order" (TSO) as our memory consistency model. TSO is a mainstream practical weak memory consistency model followed by the x86 architectures (both Intel and AMD). TSO is a more relaxed consistency model compared with strict consistency (SC). By integrating the store buffer, store and load instructions can be reordered. Stores in the store buffer follows the FIFO ordering and such ordering remains unchanged. Thus, we say the stores become visible in the cache with the same order that they are issued.

5.2 Durable Libraries Implementation Routine

This paper discusses the legality by using durable queue implementations as the case study. However, while developing the algorithm, we discover that all durable library implementations adhere to a certain *routine* regardless the type of data structure. They are: (1) execute a *newStrand*, (2) store the metadata of the concurrent object to track the current progress of the executing thread, (3) execute a *pbarrier* to restrict the ordering constraint inside the strand, (4) update the value of concurrent object by carrying out the library operation, (5) execute a *joinStrand* at the end of the library implementation. Note that (1) $\sim$ (4) is repeatable if a library implementation has more than one instruction to update the concurrent object. For instance, in an enqueue/dequeue implementation, the update of node and pointer can be put into two different strands of a thread even they have dependency. This is because these two updates read/write the same location and the persist ordering of them must respect the store ordering. Such a constraint can be explicitly shown by the behavior of the strand persistency model. The *joinStrand* in the step (5) makes sure the effect of the library operation gets response before the end of the current call. Other concurrent objects such as tree and linked-list can also adapt such a routine. Following the steps above, the persist of log in each strand can be parallelized without destroying the consistency of a concurrent object status in PM.

Note that the implementation routine shown in this paper is different with the one in Ref. [6] where each store has a corresponding undo-log, *pbarrier*, and *newStrand*. We exploited a strand in a more coarse-grain granularity and allow each strand contains more than one stores. Besides, the undo-log is created for each concurrent object but not for each store. An example of the implementation is shown in Algorithms 2 $\sim$ 5 in Section 5.5.

5.3 Buffered Durable Linearizability [11]

Buffered durable linearizability is the correctness showing that operations become persistent before they return. When a crash happens, all previously completed operations remain completed, with their effects visible in PM, or becoming durable. If operations have not completed when a crash happens, or a crash interrupts the execution, their effect may not be visible after the recovery in the subsequent history. Operations may become durable simply because they have finished all the persists before the crash even the lack of a response. Assume a history H has the happens-before order $\prec_h$ of events E, a "$\prec$-consistent cut" of H is denoted as P where if $E \in P$ and $E' \prec_h E$ in H, then $E' \in P$ and $E' \prec_h E$ in P. We use ε_i to show the ith sub-history of H without any crash event, and c_i to show the ith crash event. Then, the buffered durable linearizability is defined as follows:

Definition 3 (Buffered durable linearizability). *History* ($H = \varepsilon_0 c_1 \varepsilon_1 c_2 \ldots c_c \varepsilon_c$) *is buffered durable linearizable if there exists sub-histories* $P_0 P_1, \ldots, P_{c-1}$ *such that* $\forall 0 \leq i \leq c$, P_i *is a* $\prec$*-consistent cut of* ε_i*, and* $P_0 P_1 \ldots P_{i-1} \varepsilon_i$ *is linearizable.*

5.4 Correctness Condition

We adopt several basic axiomatic concepts from the verification of the linearizability and buffered durable linearizability to define the criteria of the strand library implementation. The basic idea is similar to the correctness condition of linearizability, which says an implementation is correct with respect to the specification of *ABS* if for every history H in the implementation, $H|ABS$ is linearizable. In other words, for every possible linearized value r in the concrete implementation history $H|REP$, there exists an abstract function that shows r respects the rep invariant I and such values are included in a possible linearized value set from $H|ABS$. We extend this idea into buffered durable linearizability in this section.

5.4.1 Rep Operations and Operational Semantics

Except for the general REP operations, memory persistency primitives should also be recognized as REP operations and can be included in a history H or $H|REP$. These primitives are not related to the rep value in the cache directly. In contrast, they manipulate the memory status and a rep value in PM and some of them are recorded in the log. We can use the operational semantics in Section 3 to capture how these primitives modify a concurrent object in PM.

5.4.2 Correctness of Durable Implementation

The central correctness condition relies on a rep invariant and an abstract function of a durable implementation under the base memory persistency model. As well as linearizability, a concurrent object needs to keep the rep invariant be true in a current history when a system is stable. Whenever a system crashes, the rep invariant must be true in the recovered history. Besides, an abstract function also needs to get the value of the object in the recovered history, which we call "durable linearized value", that satisfies the rep invariant. As described in Section 4.2, $H|REP$ is linearizable in default because the concurrent execution to the same memory location is serialized in instruction level. It is required to show that every recovered history of concurrent object r is always buffered durable linearizable. That means the implementation should keep the invariant unchanged over abstract invocations and responses, and over completions of rep operations. If any possible state violates the invariant, the implementation is incorrect.

Since a library implementation is data structure specific, the rep invariant is also an ad hoc property. However, as we discussed in Section 5.2, durable library implementations follow a certain design routine. This means that apart from the data structure specific rep invariant, a durable implementation must also satisfy the invariant in the aspect of logging and *persistent point*. For the implementation based on undo-logging, the *persistent point* refers to the instructions that perform an in-place update of a concurrent object in PM. Here, we list two invariants of persistent point:

- (A) If an in-place update of a value is shown in the current PM status, the corresponding persist of the undo log must be included in the recovered history.
- (B) A linearization point of an implementation must appear before the persistent point of the implementation.

As a system crash happens in non-deterministic way, it is not sufficient to merely capture the linearized values of the current $H|REP$. Instead, the linearized values of recovered history after every possible crashes must be considered. We use $H = \varepsilon_0 c_1 \varepsilon_1 \dots c_c \varepsilon_c$ to denote the history with c crash events. Each ε_i is a sub-history of H without any crash, c_i means the ith crash. When a system crashes at c_{i+1}, the recovery process will determine which part of ε_i should be recovered by checking the current statues of PM and ε_i. The recovered history of ε_i is denoted as P_i. Note that P_i is similar with the consistent-cut of ε_i defined in Section 5.3. Since P_i is known as the recovered history of ε_i, it is obvious to know that $P_i \subseteq \varepsilon_{i+1}$ because ε_{i+1}include both the recovered history of ε_i and ongoing history of the program.

From the definition of durable linearizability or buffered durable linearizability, P_i is another form of $trunc(comp(\varepsilon_i))$ where either adding a corresponding response or removing a pending invocation depends on whether a persistent point of the implementation is persisted or not, respectively. Here, we can obtain P_i by continuously inspecting the current memory status in PM. As we use undo-logging mechanism for recovery, ϵ_i has a pairwise persists of undo log and corresponding in-place update for each store. If a system crashes after the persist of the undo log and before the in-place update, the pending invocation is removed. If a system crashes after the persist of the in-place update and before the response of the abstract operation, a corresponding response is added.

We use $Lin(H)$ and $DurLin(H)$ to denote the sets of all linearized values of H in the cache and durable linearized value of H in PM, respectively. Similar to a linearized value, we call the memory status in PM at the end of a linearization of H as *linearized memory status* and use $DurPM(H)$ to denote the set of all possible linearized memory status in PM. Now, we can define the correctness of a durable implementation more precisely based on the invariant and buffered durable linearizability:

Definition 4 (Correctness of durable implementation). *For history $H = \varepsilon_0 c_1 \varepsilon_1 \dots c_c \varepsilon_c$, we say H is buffered durable linearizable if $\forall i.\ 0 < i < c$, P_i is a recovered history of a sub-history ε_i with PM status $DurPM(\varepsilon_i)$ and $r \in DurLin(P_i|REP)$. r holds the invariant of the persistent point I_p and the invariant of data structure I_s, and $r \in DurLin(\varepsilon_i|ABS)$.*

Note that the above correctness is independent from the data structure. To adapt such a criteria into a library implementation, a definition of a data structure specific invariant I_s is required.

5.4.3 The Invariant of Durable Queue Implementation

To obtain the adequate invariant and abstract function of the queue, we seek for a connection between the buffered durable linearized value and the recovered history. We treat a set of values in a durable queue persisted in PM as an array with partial order $\prec_p$, named Q_p. A set of values in a durable queue stayed in the cache is treated as an array with partial order $\prec_v$, named Q_v. In the recovered history, we say $x \prec_h y$ if an in-place persist of the value x at the memory location l_1 and that of the value y at the memory location l_2 has the following relation:

$$\forall H, \langle P, M, H_0\rangle \xrightarrow{*} \langle P', M', H\rangle, pwb(l_1) \prec_h pwb(l_2)$$

where H_0 means the initial empty history, and $\xrightarrow{*}$ stands for the transition from the initial state to the final state. We define the linearization point ($lp(\tau, \mu)$) as the last *st* that modifies a concur-

rent object in the cache by the abstract operation μ in the thread τ. Similarly, we also define the persistent point ($pp(\tau, \mu)$) as the last *pwb* that modifies the object in PM. If the thread τ performs enq(queue, x) in μ and the thread τ' performs enq(queue, y) in μ', we define the partial orders in Q_p and Q_v as follows, respectively:

- $x \prec_p y$ in Q_p if $pp(\tau, \mu) \prec_h pp(\tau', \mu')$ in H and $x, y \in$ *queue* in PM
- $x \prec_v y$ in Q_v if $lp(\tau, \mu) \prec_h lp(\tau', \mu')$ in H and $x, y \in$ *queue* in the cache.

Then, the abstract function shows that the durable linearized values $DurLin(H)$ in the queue in PM makes an array satisfying:

- (1) All in-place persist of values in the recovered history must also appear in the array.
- (2) if a partial order exists between two in-place persist in the recovered history, then such a partial order must also be satisfied in the array.

As a result, the invariant I_s of the durable implementation of the queue must meets the specification of the queue:

- (C) If the queue has items in it, then its length is bigger than 1.
- (D) The pointer points its next element in the queue is null for the last element.
- (E) if $x \prec_p y$ in the durable linearized values in PM, the $x \prec_v y$ in the linearized values in the cache.

5.5 Discussion of Durable Queue Implementations

In this section, we discuss the case studies of two enqueue/dequeue implementations using the strand persistency model. The case studies in this section are based on the algorithms that has been guaranteed to be correct in the aspect of memory consistency. We transformed these implementations using the strand persistency model to meet the memory persistency.

We first made an example of enqueue/dequeue implementation using *CAS* to support the durability and recoverability (Algorithms 2 and 3).

Algorithm 2 is an extension of the traditional MS queue in Algorithm 1, which uses the implementation routine described in Section 5.2. This implementation is safe in both memory consistency and persistency. In the aspect of memory persistency, this program has two strands. To abbreviate the discussion, we named the strand from L_1 to L_{11} as "data-strand", L_{13} to L_{43} as "ptr-strand". In the data-strand, right after the node is allocated, the data_log is initialized and persisted with the flag "unallocated". After the in-place update of the node, the flag is marked as "invalid" showing that the recovery of the data-strand is not required anymore. In the ptr–strand, the initial flag of ptr_log is "valid". After the in-place update of both *&tail* and &(*last* $\rightarrow$ *next*), the ptr-strand does not need to recovery when a system crashes. When the flags in both the data-strand and the ptr-strand are marked as "invalid" and such flags have reached PM, the library operation is recognized as "complete" and need to roll-forward when a system crashes. If any of the flag in PM is still "valid" or "unallocated", the library operation is required to roll-back when a system crashes even if one of the strands has completed.

Note that both data_log and ptr_log along with their flags are local to each thread so that the protection of the atomicity of log is not required. When the system recovery from a crash, a consistent queue can be recovered by comparing each log of its associated thread with the queue in PM. Such a recovery mechanism may sacrifice performance in recoverability as we re-do some of strands. However, as a system crash is known as a rare condition, we prefer to pay more attention to gain better performance when a system does not crash, which means we use the strand primitives as less as we can protect the least recoverability of the program. Since the scope of this paper focuses on the correctness of the memory persistency, the algorithm to maximize the performance of a library implementation is not discussed here.

We now discuss the memory persistency of this algorithm is safe in following three aspects: "intra-thread/intra-strand", "intra-thread/inter-strand" and "inter-thread/inter-strand".

From the view of intra-thread/intra-strand, the *pbarrier*s in L_6

Algorithm 2: Enqueue for durable MSQ with CAS

```
   Data: value
 1 newStrand;
 2 Node* node = (Node*)malloc(sizeof(Node));
 3 data_log.data = node;
 4 data_log.flg = unallocated;
 5 pwb data_log;
 6 pbarrier;
 7 node.data = value;
 8 pwb node;
 9 pbarrier;
10 data_log.flg = invalid;
11 pwb data_log.flg;
12
13 newStrand;
14 ptr_log.next = tail->next;
15 ptr_log.tail = tail;
16 ptr_log.flg = valid;
17 pwb ptr_log;
18 pbarrier;
19 while true do
20     Node* last = tail;
21     Node** nxt = &(last -> next);
22     if last == tail then
23         if nxt == NULL then
24             if CAS(&(last->next), *nxt, node) then
25                 pwb &(last->next);
26                 if CAS(&tail, *last, node) then
27                     pwb &tail;
28                 end
29                 pbarrier;
30                 ptr_log.flg = invalid;
31                 pwb ptr_log.flg;
32                 joinStrand;
33                 return;
34             end
35         else
36             pwb &(last->next);
37             if CAS(&tail, *last, *nxt) then
38                 pwb &tail;
39             end
40             pbarrier;
41         end
42     end
43 end
```

Algorithm 3: Dequeue for durable MSQ with CAS

```
1  newStrand;
2  ptr_log.tail = tail;
3  ptr_log.head = head;
4  ptr_log.flg = valid;
5  pwb ptr_log;
6  pbarrier;
7  while true do
8      Node* first = head;
9      Node* last = tail;
10     Node** nxt = &(first -> next);
11     if first == head then
12         if first == last then
13             if next == NULL then
14                 return;
15             end
16             if CAS(&tail, *last, *nxt) then
17                 pwb &tail;
18             end
19         else
20             if CAS(&head, *head, *nxt) then
21                 pwb &head;
22                 ptr_log.flg = invalid ;
23                 pwb ptr_log.flg;
24                 joinStrand;
25                 return;
26             end
27         end
28     end
29 end
```

and L_9 strictly restrict the persist ordering of the data-strand under the undo-logging mechanism, which shows that the in-place update of a memory location must be processed after the log is persisted, with the help of rules $R.3, R.4$ and $R.8, R.9, R.10$ in Section 3. Note that $R.4$ and $R.10$ indicate that L_8 is allowed to be issued before L_5 completes and after L_5 is issued, whereas L_8 is guaranteed to completes after L_5 completes. Note that the issue order of L_8 and L_5 to PM by a strand buffer is strictly protected by $R.10$, as long as a persisting table holds pending persists, the upcoming persists after the *pbarrier* in a persist buffer have to be stalled. The *pbarrier*s of L_{29} and L_{40} in the ptr-strand work in the same way. Besides, it is also allowed that a thread executes several enqueue continuously. In this case, the *joinStrand* in L_{32} restricts the order of the *pwb* and the *pbarrier* from different operations. From the rule in $R.10$, when a thread executes a *joinStrand*, it needs to make sure that all the running memory propagation are finished. In Algorithm 2, as the *joinStrand* is executed right before the return of the operation, none of *pwb* in the following operation can be issued until all the *pwb*s in the previous operation become durable in PM.

From the view of intra-thread/inter-strand, all the *pwb*s in the ptr-strand are allowed to be issued without stalling or waiting *pwb*s in the data-strand completion. This may results in an invalid persist ordering between L_8 in the data-strand and L_{25} and L_{27}, or L_{36} and L_{38} in the ptr-strand. For example, L_{25} is issued after L_8 is issued but L_{25} completes before L_8 completes. As *joinStrand* is the only operation ensuring the persist ordering of two *pwb*s in two different strands, if they are writing to different memory locations, we allow such an invalid persist ordering temporary. However, as discussed before, if a system crashes in this phase, the operation is recognized as incomplete and is required to roleback and re-do the whole operation again. This means the undo-logging mechanism in this implementation prevent the program to take reference from an incomplete memory status in PM during recovery. An alternative way to force the persist ordering of *pwb*s in the data-starnd and the ptr-strand is to put a *joinStrand* at the beginning of the ptr-strand (between L_{13} and L_{14}). However, it results in that the ptr-strand must stall until the data-strand completes, which sacrifices the performance instead.

From the view of inter-thread/inter-strand, the ordering constraint of L_{25}, L_{27}, L_{31}, L_{36}, and L_{38} are protected by how *CAS* keeps the atomicity of queue update. L_{25}, L_{27}, and L_{31} are issued only when L_{24} is triggered and the *joinStrand* in L_{32} makes sure these three *pwb*s need to complete before the end of the implementation. Such an implementation makes sure the update of the ptr-strand is atomic and cannot be interrupted by other threads. L_{36} and L_{38} are added into the current ptr-strand when L_{23} is false and the *pbarrier* in L_{40} restricts the persist ordering of the *pwb* in the ptr-strand between the current loop and the next loop. As a result, the durable enqueue is legal in the durability with the formalized behavior of the strand persistency model.

From the view of recoverability, when a system crashes, the recovery process traverses the queue in the current PM from the head to the tail. The traverse finishes at the tail with "invalid" flag is located. Such a traverse is used to locate the tail where the enqueue for the tail is interrupted. If the flag of the corresponding enqueue of the tail is marked as "valid/unallocated", the enqueue is required to be executed again.

The legality of the dequeue in Algorithm 3 can be discussed in the same way. The implementation of the dequeue only contains the ptr-strand so that the case of intra-thread/inter-strand does not have to be considered. As same as the enqueue, the atomicity of *pwb*s in L_{21} and L_{23} are guaranteed by *CAS* in L_{20}. The *joinStrand* in L_{24} makes sure all effects of the persistent operations must get response before the end of the dequeue operation. The recovery of the dequeue is similar with the enqueue, a traverse is processed from the head to the tail, and finish the traverse at the head with "invalid" flag is located. All the dequeues with "invalid" flag are required to be executed again.

We made an another example of a durable strand lock-based enqueue/dequeue implementation (Algorithms 4 and 5). As same as MS queue, a new data is allocated in the data segment first, then it updates the pointer of the queue to let the tail point to the data segment. The durability of the enqueue and the dequeue can be protected in the same way. We use the enqueue to explain the mechanism behind it.

From the view of memory consistency, this program is safe in a concurrent environment. When a thread performs the enqueue, a node is allocated in the local environment. Such allocation is independent from other threads so that it can be concurrently performed with other threads. When the thread updates the queue, the critical section is protected by lock to keep all the manipulations are atomic in the aspect of inter-thread. Besides, the pointer manipulation of the thread can be concurrent with the data allocation from other threads. The data allocation must be performed

Algorithm 4: Enqueue for durable queue with lock

```
   Data: value
1  newStrand;
2  Node* node = (Node*)malloc(sizeof(Node));
3  data_log.data = node;
4  data_log.flg = unallocated;
5  pwb data_log;
6  pbarrier;
7  node.data = value;
8  pwb node;
9  pbarrier;
10 data_log.flg = invalid;
11 pwb data_log.flg;
12
13 lock(ptr_lock);
14 fence;
15 newStrand;
16 ptr_log.next = tail;
17 ptr_log.tail = tail;
18 ptr_log.flg = valid;
19 pwb ptr_log;
20 pbarrier;
21 tail->next = node;
22 pwb &(tail->next);
23 tail = node;
24 pwb &tail;
25 pbarrier;
26 ptr_log.flg = invalid;
27 pwb ptr_log.flg;
28 joinStrand;
29 unlock(ptr_lock);
30 return;
```

Algorithm 5: Dequeue for durable queue with lock

```
1  lock(ptr_lock);
2  fence;
3  newStrand;
4  ptr_log.head = head;
5  ptr_log.flg = valid;
6  pwb ptr_log;
7  pbarrier;
8  Node* first = head;
9  Node* nxt = head->next;
10 if first == NULL then
11     ptr_log.flg = invalid;
12     pwb ptr_log.flg;
13     joinStrand;
14     unlock(ptr_lock);
15     return;
16 end
17 head = nxt;
18 pwb head;
19 pbarrier;
20 ptr_log.flg = invalid;
21 pwb ptr_log.flg;
22 joinStrand;
23 unlock(ptr_lock);
24 return;
```

before the pointer manipulation in intra-thread. Such a restricted ordering is also protected by the "fence" after the lock.

In the aspect of memory persistency, the data-strand is from L_1 to L_{11} and the ptr-strand is from L_{13} to L_{29}. The behavior in the data-strand is same as that in Algorithms 2. In the ptr-strand, the log of the queue is initialized right after the lock. After the log is persisted, two pointers are persisted separately. Note that no persist ordering between persists of these two pointers exists as failure atomicity only allow none or all of persists being durable. As well as the data-strand, the upcoming persist of flag means the log is invalid anymore. Finally, before the unlock, the *joinStrand* is performed to make sure all the stores are visible and all the persists are durable.

All above algorithms meet the correctness criteria of Definition 4. First, $DurLin(P_i|REP) \subseteq DurLin(\varepsilon_i|ABS)$ is always true because the library operation is able to rollback to a certain stable stage with the help of undo-logging. Next, $DurPM(\varepsilon_i)$ can be obtained by following the trace of *pwb*s in the current history ε_i. From ε_i, the invariant of the persistent point and the queue are always met because both CAS and *lock* can implicitly protect the persistent ordering of both inter-thread/inter-strand and intra-thread/inter-strand. This is because the execution of each *pwb* can be triggered only when the corresponding CAS return true, or *lock* is activated. Thus, the order of *pwb*s follows the order of CASs or the order of each thread acquires *lock*.

In **Fig. 3**, we use an example of proof script to show the mechanism behind it. The precondition of the proof is as follows: a queue Q is provided with the value $[a; b]$. Two threads attempts to perform $Enq(Q, c)$ and $Enq(Q, d)$ concurrently. The Hypothesis is when both of two operations complete, the Q in PM is $[a; b; c]$ where the *tail* of Q is c and the $tail \rightarrow next$ of Q is $NULL$. The goal of proof is showing such hypothesis is false.

From H, we know that both the nodes of c and d have been added into the queue in order, but the tail is failed to be updated. This means the effect of $T_1.pwb(\&tail)$ overwrites the effect of $T_2.pwb(\&tail)$, whereas the effect of $T_2.pwb(\&(last \rightarrow next))$ overwrites the effect of $T_1.pwb(\&(last \rightarrow next))$. It decides the completion order of these four instructions, which is shown in S_2. As the issue order of *pwb*s is same with the completion order of these *pwb*s if they are writing back from the same location in the cache, we can infer the issue order in S_4 from S_3. Because after the issue of $pwb(\&tail)$ and $pwb(\&(last \rightarrow next))$, the PM status is changed, we can infer that after T_2 issued $pwb(\&tail)$, T_1 has modified the value of $\&tail$ in the cache by CAS, and after T_1 issued $pwb(\&(last \rightarrow next))$, T_2 has modified the value of $\&(last \rightarrow next)$ in the cache by CAS. This is shown in S_6. Next, from the behavior of a *pwb*, we know that before any *pwb* can be issued, the corresponding CAS has to be visible in the cache. Thus, we can get the order of the CAS shown in S_7. From Algorithm 2, we know that the atomicity of the enqueue is protected by the CAS. The update of $\&tail$ and $\&(last \rightarrow next)$ by a thread cannot be interrupted by another thread. Therefore, it is impossible to detect the ordering that the $T_1.L_{24} \prec T_2.L_{24}$ whereas $T_2.L_{26} \prec T_1.L_{26}$, which is shown from S_8 to S_{11}. As a result, we proved that the ordering in S_7 is impossible. Thus, the hypothesis is incorrect.

Precondition:
P_1: $T_1.enq(Q,c)||\ T_2.enq(Q,d)$
P_2: $Q = \{a,b\}$

Hypothesis:
$\mathcal{H}$: $(T_1.enq(Q,c).\text{compl} = \text{true}) \wedge (\text{T}_2.\text{enq}(\text{Q},\text{d}).\text{compl} = \text{true})$
$\rightarrow\ (b.next = c) \wedge (c.next = d) \wedge (d.next = NULL) \wedge (Q.tail\ =\ c)$

Proof:
S_1: $\because \mathcal{H}$
S_2: $\therefore\ T_2.pwb(\&tail).comp \prec T_1.pwb(\&tail).comp,$
$T_1.pwb(\&last \rightarrow next).comp \prec T_2.pwb(\&last \rightarrow next).comp$

S_3: $\because\ T_i.pwb(loc).comp \prec T_j.pwb(loc).comp\ \Longrightarrow T_i.pwb(loc).issue \prec T_j.pwb(loc).issue$
S_4: $\therefore\ T_2.pwb(\&tail).issue \prec T_1.pwb(\&tail).issue,$
$T_1.pwb(\&last \rightarrow next).issue \prec T_2.pwb(\&last \rightarrow next).issue$

S_5: $\because\ T_2.pwb(\&tail).issue \prec T_1.pwb(\&tail).issue$
$\Longrightarrow PM(\&tail_{T2}) \neq PM(\&tail_{T1})$
$T_1.pwb(\&last \rightarrow next).issue \prec T_j.pwb(\&last \rightarrow next).issue$
$\Longrightarrow PM\big((\&last \rightarrow next)_{T1}\big) \neq PM((\&last \rightarrow next)_{T2})$
S_6: $\therefore\ T_2.pwb(\&tail).issue \prec T_1.CAS(\&tail) \prec T_1.pwb(\&tail).issue,$
$T_1.pwb(\&last \rightarrow next).issue \prec T_2.CAS(\&last \rightarrow next) \prec T_2.pwb(\&last \rightarrow next).issue$
S_7: $\therefore\ T_2.CAS(\&tail) \prec T_1.CAS(\&tail),$
$T_1.CAS(\&last \rightarrow next) \prec T_2.CAS(\&last \rightarrow next)$

S_8: $\because \{T_1.L_{24} \prec T_2.L_{24}\} \wedge \{T_1.L_{24} \prec T_1.L_{26}\}$
S_9: $\therefore \{T_2.L_{23} = FALSE\}$
S_{10}: $\therefore \{T_2.L_{26} \prec T_1.L_{26}\}$
S_{11}: $\therefore\ \{T_2.CAS(\&tail) \prec T_1.CAS(\&tail\}) = FALSE$

S_{12}: $\therefore \mathcal{H} = FALSE$
S_{13}: $\therefore \{Assumption\ is\ incorrect\}$

Fig. 3 The proof script of impossible PM state and memory ordering.

5.6 Related Work of Durable Queue Implementations

The design of durable data structures that can recover from system crashes has been studied in the past years. In Friedman et al. work [5], they proposed three novel implementations of a concurrent lock-free queue in high-level design. As same as the implementations in this paper, they extended the traditional Michael and Scott's queue to meet the memory persistency model. The three implementations proposed in their work are different in the level of durable linearization, which also refers to the permitted behaviors in the aspect of memory persistency. The proposed implementations in this paper are different with their work, especially in the aspect of memory persistency model and concurrency. In Friedman et al. work, they use *FLUSH* instruction, which is similar with the *pwb* in this paper, to perform the write-back from the cache to PM. Since *FLUSH* takes time to get response, the durable implementations in their work optimize the program by minimizing the number of *FLUSH*s, whereas guarantee the least durability and recoverability of the program. The concurrency among *FLUSH*s are not included in the scope of their work so that all the execution of *FLUSH*s are sequential. Besides, since the algorithms are implemented in high-level, the low-level memory persistency models are not discussed in their work.

In the contrast, the proposed implementations in this paper focus on the concurrency among write-back operations in a durable implementation, and study the legality of a durable implementation while using the most relaxed low-level memory persistency model. As a result, in our implementation, as long as the durability and recoverability of a program are not destroyed, it is possible to perform the write-back to PM without stalling until the previous write-back is complete.

6. Conclusion

In this paper, we first formally design the operational semantics of the strand persistency model to formalize the behavior of it. The semantics describes how a history, a storage, and a program are changed under the strand primitives. We then proposed the criteria that describe the durability and recoverability of a strand program by extending the buffered durable linearizability. Finally, we use two queue implementations as case studies to discuss the legality of durable library implementations.

Acknowledgments We would like to offer our special thanks to professor Tachio Terauchi from Waseda university for helpful discussions and comments about the theoretical part of this paper.

References

[1] Chakrabarti, D.R., Boehm, H.-J. and Bhandari, K.: Atlas: Leveraging locks for non-volatile memory consistency, *ACM SIGPLAN Notices*, Vol.49, No.10, pp.433–452 (2014).
[2] Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R. and Swanson, S.: NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories, *ACM SIGARCH Computer Architecture News*, Vol.39, No.1, pp.105–118 (2011).
[3] Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D. and Coetzee, D.: Better I/O through byte-addressable, persistent memory, *Proc. ACM SIGOPS 22nd symposium on Operating systems principles*, pp.133–146 (2009).
[4] Derrick, J., Doherty, S., Dongol, B., Schellhorn, G. and Wehrheim, H.: Verifying correctness of persistent concurrent data structures, *International Symposium on Formal Methods*, pp.179–195, Springer (2019).
[5] Friedman, M., Herlihy, M., Marathe, V. and Petrank, E.: A persistent lock-free queue for non-volatile memory, *ACM SIGPLAN Notices*, Vol.53, No.1, pp.28–40 (2018).
[6] Gogte, V., Wang, W., Diestelhorst, S., Chen, P.M., Narayanasamy, S. and Wenisch, T.F.: Relaxed Persist Ordering Using Strand Persistency, *2020 ACM/IEEE 47th Annual International Symposium on Computer*

Architecture (*ISCA*), pp.652–665 (online), DOI: 10.1109/ISCA45697.2020.00060 (2020).
[7] Gogte, V., Diestelhorst, S., Wang, W., Narayanasamy, S., Chen, P.M. and Wenisch, T.F.: Persistency for Synchronization-Free Regions, *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.46–61, Association for Computing Machinery (online), DOI: 10.1145/3192366.3192367 (2018).
[8] Herlihy, M.P. and Wing, J.M.: Linearizability: A correctness condition for concurrent objects, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.12, No.3, pp.463–492 (1990).
[9] Intel: Intel® Architecture Instruction Set Extensions and Future Features Programming Reference (2020), available from ⟨https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf⟩.
[10] Izraelevitz, J., Kelly, T. and Kolli, A.: Failure-Atomic Persistent Memory Updates via JUSTDO Logging, *ASPLOS '16*, pp.427–442, Association for Computing Machinery (online), DOI: 10.1145/2872362.2872410 (2016).
[11] Izraelevitz, J., Mendes, H. and Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model, *International Symposium on Distributed Computing*, pp.313–327, Springer (2016).
[12] Joshi, A., Nagarajan, V., Cintra, M. and Viglas, S.: Efficient persist barriers for multicores, *Proc. 48th International Symposium on Microarchitecture*, pp.660–671 (2015).
[13] Kolli, A., Gogte, V., Saidi, A., Diestelhorst, S., Chen, P.M., Narayanasamy, S. and Wenisch, T.F.: Language-level persistency, *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture* (*ISCA*), pp.481–493 (online), DOI: 10.1145/3079856.3080229 (2017).
[14] Kolli, A., Pelley, S., Saidi, A., Chen, P.M. and Wenisch, T.F.: High-performance transactions for persistent memories, *Proc. 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.399–411 (2016).
[15] Lee, B.C., Ipek, E., Mutlu, O. and Burger, D.: Architecting phase change memory as a scalable dram alternative, *Proc. 36th Annual International Symposium on Computer Architecture*, pp.2–13 (2009).
[16] Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W. and Ren, J.: DudeTM: Building durable transactions with decoupling for persistent memory, *ACM SIGPLAN Notices*, Vol.52, No.4, pp.329–343 (2017).
[17] Liu, Q., Izraelevitz, J., Lee, S.K., Scott, M.L., Noh, S.H. and Jung, C.: iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory, *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), pp.258–270 (online), DOI: 10.1109/MICRO.2018.00029 (2018).
[18] Liu, S., Wei, Y., Zhao, J., Kolli, A. and Khan, S.: PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs, *Proc. 24th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, pp.411–425, Association for Computing Machinery (online), DOI: 10.1145/3297858.3304015 (2019).
[19] Michael, M.M. and Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, pp.267–275 (1996).
[20] Nalli, S., Haria, S., Hill, M.D., Swift, M.M., Volos, H. and Keeton, K.: An analysis of persistent memory use with WHISPER, *ACM SIGPLAN Notices*, Vol.52, No.4, pp.135–148 (2017).
[21] Ogleari, M.A., Miller, E.L. and Zhao, J.: Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems, *2018 IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), pp.336–349 (online), DOI: 10.1109/HPCA.2018.00037 (2018).
[22] Pelley, S., Chen, P.M. and Wenisch, T.F.: Memory persistency, *2014 ACM/IEEE 41st International Symposium on Computer Architecture* (*ISCA*), pp.265–276, IEEE (2014).
[23] Qureshi, M.K., Srinivasan, V. and Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology, *Proc. 36th Annual International Symposium on Computer Architecture*, pp.24–33 (2009).
[24] Raad, A. and Vafeiadis, V.: Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model, *Proc. ACM Program. Lang.*, Vol.2, No.OOPSLA, p.27 (online), DOI: 10.1145/3276507 (2018).
[25] Raad, A., Wickerson, J., Neiger, G. and Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture, *Proc. ACM on Programming Languages*, Vol.4, No.POPL, pp.1–31 (2019).
[26] Schellhorn, G. and Wehrheim, H.: Defining and Verifying Durable Opacity: Correctness for Persistent Software Transactional Memory, *Formal Techniques for Distributed Objects, Components, and Systems*, pp.39–58, Springer International Publishing (2020).
[27] Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z. and Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors, *Comm. ACM*, Vol.53, No.7, pp.89–97 (2010).
[28] Strukov, D.B., Snider, G.S., Stewart, D.R. and Williams, R.S.: The missing memristor found, *Nature*, Vol.453, No.7191, pp.80–83 (2008).
[29] Volos, H., Tack, A.J. and Swift, M.M.: Mnemosyne: Lightweight persistent memory, *ACM SIGARCH Computer Architecture News*, Vol.39, No.1, pp.91–104 (2011).

Jixin Han was born in 1992. He received his M.S. degree from Waseda University in 2017. He was a research assistant in Waseda University from 2017 to 2020. He was one of the founder of IEEE-HKN Japanese Chapter in 2018, wan president of IEEE-HKN Japanese Chapter from 2019 to 2020. He is currently a member of IEEE. His research interest is formal verification, compilation and program optimization.

Keiji Kimura received his Ph.D. degrees in electrical engineering from Waseda University in 2001. He was an assistant professor in 2004, associate professor in 2005, and professor in 2012 at Waseda University. He is a director of Green Computing System Research Organization in Waseda from 2019. He is a recipient of 2014 MEXT (Ministry of Education, Culture, Sports, Science and Technology in Japan) award. His research interest includes multicore processor architecture and parallelizing compiler technologies. He is a member of IPSJ, ACM and IEEE. He has served on program committee of many conferences.