The Fence Complexity of Persistent Sets

Gaetano Coccimiglio[1], Trevor Brown[2], and Srivatsan Ravi[3]

[1] University of Waterloo, Waterloo ON N2L 3G1, Canada
gccoccim@uwaterloo.ca
[2] University of Waterloo, Waterloo ON N2L 3G1, Canada
trevor.brown@uwaterloo.ca
[3] University of Southern California, Los Angeles CA 90007, USA
srivatsr@usc.edu

Abstract. We study the psync complexity of concurrent sets in the non-volatile shared memory model. Flush instructions are used in non-volatile memory to force shared state to be written back to non-volatile memory and must typically be accompanied by the use of expensive fence instructions to enforce ordering among such flushes. Collectively we refer to a flush and a fence as a psync. The safety property of strict linearizability forces crashed operations to take effect before the crash or not take effect at all; the weaker property of durable linearizability enforces this requirement only for operations that have completed prior to the crash event. We consider lock-free implementations of list-based sets and prove two lower bounds. We prove that for any durable linearizable lock-free set there must exist an execution where some process must perform at least one redundant psync as part of an update operation. We introduce an extension to strict linearizability specialized for persistent sets that we call strict limited effect (SLE) linearizability. SLE linearizability explicitly ensures that operations do not take effect after a crash which better reflects the original intentions of strict linearizability. We show that it is impossible to implement SLE linearizable lock-free sets in which read-only (or search) operations do not flush or fence. We undertake an empirical study of persistent sets that examines various algorithmic design techniques and the impact of flush instructions in practice. We present concurrent set algorithms that provide matching upper bounds and rigorously evaluate them against existing persistent sets to expose the impact of algorithmic design and safety properties on psync complexity in practice as well as the cost of recovering the data structure following a system crash.

Keywords: Strict linearizability · Durable linearizability · Lower bounds · Persistent sets · Non-volatile memory.

1 Introduction

Byte-addressable Non-Volatile Memory (NVM) is now commercially available, thus accelerating the need for efficient *persistent* concurrent data structure algorithms. We consider a model in which systems can experience full system crashes.

When a crash occurs the contents of volatile memory are lost but the contents of NVM remain persistent. Following a crash a recovery procedure is used to bring the data structure back to a consistent state using the contents of NVM. In order to force shared state to be written back to NVM the programmer is sometimes required to explicitly *flush* shared objects to NVM by using *explicit flush* and *persistence fence* primitives, the combination of which is referred to as a *psync* [21]. While concurrent sets have been extensively studied for volatile shared memory [14], they are still relatively nascent in non-volatile shared memory. This paper presents a detailed study of the psync complexity of *concurrent sets* in theory and practice.

Algorithmic design choices for persistent sets. The recent trend is to persist less data structure state to minimize the cost of writing to NVM. For example, the Link-Free and SOFT [21] persistent list-based sets do not persist any *pointers* in the data structure. Instead they persist the *keys* along with some other metadata used after a crash to determine if the key is in the data structure. This requires at most a single psync for update operations; however, not persisting the structure results in a more complicated recovery procedure.

A manuscript by Israelevitz and nine other authors presented a seminal in depth study of the performance characteristics of real NVM hardware [16]. Their results may have played a role in motivating the trend to persist as little as possible and reduce the number of fences. In particular, they found (Figure 8 of [16]) that the latency to write 256 bytes and then perform a psync is at least 3.5x the latency to write 256 bytes and perform a flush but no persistence fence. Moreover, they found that NVM's write bandwidth could be a severe bottleneck, as a write-only benchmark (Figure 9 of [16]) showed that NVM write bandwidth *scaled negatively* as the number of threads increased past four, and was approximately *9x lower* than volatile write bandwidth with 24 threads. A similar study of real NVM hardware was presented by Peng et al. [17].

While these results are compelling, it is unclear whether these latencies and bandwidth limitations are a problem for concurrent sets in practice. As it turns out, the push for *persistence-free* operations and synchronization mechanisms that minimize the amount of data persisted, and/or the number of *psyncs*, has many consequences, and the balance may favour incurring increased psyncs in some cases.

Contributions. Concurrent data structures in volatile shared memory typically satisfy the *linearizability* safety property, NVM data structures must consider the state of the persistent object following a full system crash. The safety property of *durable-linearizability* satisfies linearizability and following a crash, requires that the object state reflect a consistent operation subhistory that includes operations that had a response before the crash [15]. (i) We prove that for any durable-linearizable lock-free set there must exist an execution in which some process must perform at least one *redundant* psync as part of an update operation (§ 2). Informally, a redundant psync is one that does not change the contents of NVM. Our result is orthogonal to the lower bound of Cohen et al. who showed that the minimum number of psyncs per update for a durable-linearizable lock-

free object is one [7]. However, Cohen et al. did not consider redundant psyncs. We show that redundant psyncs cannot be completely avoided in all concurrent executions: there exists an execution where n processes are concurrently performing update operations and $n-1$ processes perform a redundant psync. (ii) Our first result also applies to *SLE linearizability*, which we define to serve as a natural extension of the safety property of *strict linearizability* specifically for persistent sets. Originally defined by Aguilera and Frølund [1], strict linearizability forces crashed operations to be linearized before the crash or not at all. Strict linearizability was not originally defined for models in which the system can recover following a crash. To better capture the intentions of strict linearizability in the context of persistent concurrent sets, we introduce SLE linearizability to realize the intuition of Aguilera and Frølund for persistent concurrent sets. SLE linearizability is defined to explicitly enforce *limited effect* for persistent sets.

(iii) We prove that it is impossible to implement SLE linearizable lock-free sets in which read-only operations neither flush nor execute persistence fences, but it is possible to implement strict linearizable and durable linearizable lock-free sets with persistence-free reads (§ 2). (iv) We study the empirical costs of persistence fences in practice. To do this, we present matching upper bounds to our lower bound contributions (i) and (ii). Specifically, we describe a new technique for implementing persistent concurrent sets with persistence-free read-only operations called the extended link-and-persist technique and we utilize this technique to implement several persistent sets (§ 3). (v) We evaluate our upper bound implementations against existing persistent sets in a systemic empirical study of persistent sets. This study exposes the impact of algorithmic design and safety properties on persistence fence complexity in practice and the cost of recovering the data structure following a crash (§ 4).

The relationship between performance, psync complexity, recovery complexity and the correctness condition is subtle, even for seemingly simple data types like sorted sets. In this paper, we delve into the details of algorithmic design choices in persistent data structures to begin to characterize their impact.

2 Lower Bounds

Persistency Model and Safety Properties . We assume a full system crash-recovery model (all processes crash together). When a crash occurs all processes are returned to their initial states. After a crash a recovery procedure is invoked, and only after that can new operations begin.

Modifications to base objects first take effect in the volatile shared memory. Such modifications become persistent only once they are flushed to NVM. Base objects in volatile memory are flushed asynchronously by the processor (without the programmer's knowledge) to NVM arbitrarily. We refer to this as a *background flush*. We consider *background flushes* to be atomic. The programmer can also *explicitly* flush base objects to NVM by invoking *flush* primitives, typically accompanied by *persistence fence* primitives. An *explicit flush* is a primitive on a base object and is non-blocking, i.e., it may return *before* the base object has

been written to persistent memory. An *explicit flush* by process p is guaranteed to have taken effect only after a subsequent *persistence fence* by p. An explicit flush combined with a persistence fence is referred to as a *psync*. We assume that psync events happen independently of RMW events and that psyncs do not change the configuration of volatile shared memory (other than updating the program counter). Note that on Intel platforms a RMW implies a fence, however, a RMW does not imply a flush before that fence, and therefore does not imply a psync.

In this paper, we consider the *set* type: an object of the set type stores a set of integer values, initially empty, and exports three operations: `insert`(v), `remove`(v), `contains`(v) where $v \in \mathbb{Z}$. A *history* is a sequence of invocations and responses of operations on the set implementation. We say a history is well-formed if no process invokes a new operation before the previous operation returns. Histories H and H' are *equivalent* if for every process p_i, $H|i = H'|i$.

A history H is durable linearizable, if it is well-formed and if $ops(H)$ is linearizable where $ops(H)$ is the subhistory of H containing no crash events [15].

Aguilera and Frølund defined strict linearizability for a model in which individual processes can crash and did not allow for recovery [1]. Berryhill et al. adapted strict linearizability for a model that allows for recovery following a system crash [2]. A history H is *strict linearizable* with respect to an object type τ if there exists a sequential history S equivalent to a *strict completion of* H, such that (1) $\rightarrow_{H^c} \subseteq \rightarrow_S$ and (2) S *is consistent with the sequential specification of* τ. A strict completion of H is obtained from H by inserting matching responses for a subset of pending operations after the operation's invocation and before the next crash event (if any), and finally removing any remaining pending operations and crash events.

Psync Complexity. It is likely that an implementation of persistent object will have many similarities to a volatile object of the same abstract data type. For this reason, when comparing implementations of persistent objects we are mostly interested in the overhead required to maintain a consistent state in persistent memory. Specifically, we consider psync complexity.

Programmers write data to persistent memory through the use of psyncs. A psync is an expensive operation. Cohen et al. [7] prove that update operations in a durable linearizable lock-free algorithm must perform at least one psync. In some implementations of persistent objects, reads also must perform psyncs. There is a clear focus in existing literature on minimizing the number of psyncs per data structure operation [9, 21, 11]. These factors suggest that psync complexity is a useful metric for comparing implementations of persistent objects.

Lower Bounds for Persistent Sets. We now present the two main lower bounds in this paper, but the full proofs are only provided in the full version of the paper[5] due to space constraints.

Impossibility of persistence-free read-only searches. The key goal of the original work of Aguilera and Frølund [1] was to enforce *limited effect* by requiring operations to take effect before the crash or not at all. Limited effect requires that an operation takes effect within a limited amount of time after it

is invoked. The point at which an operation takes effect is typically referred to as its *linearization point* and we say that the operation *linearizes* at that point. Rephrasing the intuition, when crashes can occur, limited effect requires that operations that were pending at the time of a crash linearize prior to the crash or not at all.

Strict linearizability is defined in terms of histories, which abstract away the real-time order of events. As a result, strict linearizability does not allow one to argue anything about the ordering of linearization points of operations that were pending at the time of a crash relative to the crash event. Thus, strict linearizability cannot and does not prevent operations from taking effect during the *recovery procedure* or even after the recovery procedure (which can occur for example in implementations that utilize linearization helping [4]). Strict linearizability only requires that at the time of a crash, pending operations *appear* to take effect prior to the crash. Although we are not aware of a formal proof of this, we conjecture in the full system crash-recovery model, durable linearizable objects are strict linearizable for some suitable definition of the recovery procedure. This is because we can always have the recovery procedure *clean-up* the state of the object returning it to a state such that the resulting history of any possible extension will satisfy strict linearizability. We note this conjecture as further motivation towards re-examining the way in which the definition of strict linearizability has been adapted for the full system crash-recovery model.

To this end, we define the concept of a *key write* to capture the intentions of Aguilera and Frølund in the context of sets by defining *Strict limited effect* (SLE) linearizability for sets as follows: a history satisfies SLE linearizability iff the history satisfies strict linearizability and for all operations with a key write, if the operation is pending at the time of a crash, the key write of the operation must occur before the crash event. In the strict completion of a history this is equivalent to requiring that the key write is always between the invocation and response of the operation. This is because the order of key writes relative to a crash event is fixed which means if the write occurs after the crash event then a strict completion of the history could insert a response for the operation only prior to the key write (at the crash) and this response cannot be reordered after the key write.

We show that it is impossible to implement a SLE linearizable lock-free set for which read-only searches do not perform any explicit flushes or persistence fences.

Theorem 1. *There exists no SLE linearizable lock-free set with* persistence-free *read-only searches.*

Redundant psync lower bound for durable linearizable sets. After modifying a base object only a single psync is required to ensure that it is written to persistent memory. Performing multiple psyncs on the same base object is therefore unnecessary and wasteful. We refer to these unnecessary psyncs as *redundant* psyncs. We show that for any durably linearizable lock-free set there must exist an execution in which n concurrent processes are invoking n concurrent update operations and n-1 processes each perform at least one redundant psync. At first

glance one may think that this result is implied by the lower bound of Cohen et al. [7]. Cohen et al. show that for any lock-free durable linearizable object, there exists an execution wherein every update operation performs at least one persistence fence. Cohen et al. make no claims regarding redundant psyncs. Our result demonstrates that durable linearizable lock-free objects cannot completely avoid redundant psyncs.

Theorem 2. *In an n-process system, for every durable linearizable lock-free set implementation I, there exists an execution of I wherein n processes are concurrently performing update operations and n-1 processes perform a redundant psync.*

3 Upper Bounds

Briding the gap between theory and practice. The lower bounds presented in the previous section offer insights into the theoretical limits of persistent sets for both durable linearizability and SLE linearizability. While these lower bounds demonstrate a clear separation between durable and SLE linearizability, it is unclear whether or not we can observe any meaningful separation in practice. In order to answer this question we would like to compare durable and SLE linearizable variants of the same persistent set implementation. To this end, we extended the Link-and-Persist technique [9] to allow for persistence-free searches and use our extension to implement several persistent linked-list. We also add persistence helping to SOFT [21]. We explain both in detail next.

Notable persistent set implementations. We briefly describe above mentioned existing implementations of persistent sets. We only focus on hand-crafted implementations since they generally perform better in practice compared to transforms or universal constructions [11, 12].

David et al. describe a technique for implementing durable linearizable link-based data structures called the Link-and-Persist technique [9]. Using the Link-and-Persist technique, whenever a link in the data structure is updated, a single bit mark is applied to the link which denotes that it has not been written to persistent memory. The mark is removed after the link is written to persistent memory. We refer to this mark as the *persistence bit*. This technique was also presented by Wang et al. in the same year [19]. Wei et al. presented a more general technique which does not steal bits from data structure links [20].

The Link-Free algorithm of Zuriel et al. does not persist data structure links [21]. Instead, the Link-Free algorithm persists metadata added to every node.

Zuriel et al. designed a different algorithm called SOFT (Sets with an Optimal Flushing Technique) offering persistence-free searches. The SOFT algorithm does not persist data structure links and instead persists metadata added to each node. The major difference between the Link-Free algorithm and SOFT is that SOFT uses two different representations for every key in the data structure where only one representation is is explicitly flushed to persistent memory.

Recovery complexity. After a crash, a recovery procedure is invoked to return the objects in persistent memory back to a consistent state. Prior work

has utilized a sequential recovery procedure [21, 9, 12, 8]. A sequential recovery procedure is not required for correctness but it motivates the desire for efficient recovery procedures. No new data structure operations can be invoked until the recovery procedure has completed. Ideally we would like to minimize this period of downtime represented by the execution of recovery procedure. For the upper bounds in the this section, we use the asymptotic time complexity of the recovery procedure as another metric for comparing durable linearizable algorithms.

Extended Link-and-Persist. We choose to extend the Link-and-Persist technique of David et al. because it is quite simple and it represents the state of the art for hand-crafted algorithms that persist the links of a data structure. Moreover, unlike the algorithms in [21], the Link-and-Persist technique can be used to implement persistent sets without compromising recovery complexity. We build on the Link-and-Persist technique by extending it to allow for persistence-free searches and improved practical performance. Cohen et al noted that persistence-free searches rely on the ability to linearize successful update operations at some point after the CPE of the operation [7]. In our case, this means that searches must be able to determine if the pointer is not persistent because of an `Insert` operation or a `Remove` operation. This is not possible with the original Link-and-Persist technique. We address this with two changes.

First, we require that a successful update operation, π_u, is linearized after its *Critical Persistence Event* (or CPE). Intuitively, the CPE represents the point after which the update will be recovered if a crash occurs. Specifically, if a volatile data structure would linearize π_u at the success of a RMW on a pointer v then we require that π_u is linearized at the success of the RMW that sets the persistence bit in v. If a search traverses a pointer, v, marked as not persistent the search can always be linearized prior to the concurrent update which modified v.

Secondly, since successful updates are linearized after their CPE, if the response of search operation depends on data that is linked into the data structure by a pointer marked as not persistent then the search must be able to access the last persistent value of that pointer. To achieve this, we add a pointer field to every node which we call the *old field*. A node will have both an *old field* and a pointer to its successor (*next pointer*) which effectively doubles the size of every data structure link. The *old field* will point to the last persistent value of the successor pointer while the successor pointer is marked as not persistent. In practice, the *old field* must be initialized to `null` then updated to a non-`null` value when the corresponding successor pointer is modified to a new value that needs to be persisted. Note that modifications like flagging or marking do not always need to be persisted; this depends on the whether or not the update can complete while the flagged or marked pointers are still reachable via a traversal from the root of the data structure. The easiest way to correctly update the *old field* is to update the successor pointer and the *old field* atomically using a hardware implementation of double-wide compare-and-swap (DWCAS) namely the cmpxchg16b instruction on Intel. Alternatively, a regular single-word compare-and-swap (SWCAS) can be used but this requires adding extra volatile memory synchronization to ensure correctness. For some data structures such as linked-

lists using only SWCAS might also require adding an extra psync to updates. To allow searches to distinguish between pointers that are marked as not persistent because of a `remove` versus those that are not persistent because of an `insert` we require that the *old field* is always updated to a non-`null` value whenever a `remove` operation unlinks a node. `Insert` operations that modify the data structure must flag either the *old field* or the corresponding successor to indicate that the pointer marked as not persistent was last updated by an insert. When using SWCAS to update the *old field* this flag must be on the successor pointer.

With our extension if the response of a search operation depends on data linked into the data structure via a pointer marked as not persistent it can be linearized prior to the concurrent update operation that modified the pointer and it can use the information in the *old field* to determine the correct response which does not require performing any psyncs. If the search finds that the update was an insert it simply returns `false`. If the update was a remove but the search was able to find the value that it was looking for then it can return `true` since that key will be in persistent memory. If the update was a remove but the search was not able to find the value that it was looking for then it can check the if the node pointed to by the `old field` contains the value. As with the original, our extension still requires that an operation π will ensure that the CPE of any other operation which π depends on has occurred. π must also ensure that its own CPE has occurred before it returns. Another requirement which was not explicitly stated by David et al. is that operations must ensure that any data that a data structure link can point to is written to persistent memory before the link is updated to point to that data.

Our extension can be used to implement several link-based sets including trees and hash tables. Data structures implemented using our extension provide durable linearizability, however the use of persistence-free searches is optional. If the data structure does not utilize persistence-free searches then it would provide SLE linearizability (requiring only a change in the correctness proof).

Augmenting LF and SOFT. SOFT represents the state of the art for hand-crafted algorithms that do not persist the links of a data structure. The SOFT algorithm provides durable linearizability. For comparison, we added persistence helping for all operations of a persistent linked-list implemented using SOFT (thereby removing persistence-free searches) to achieve a SLE linearizable variant. We refer to this variant as SOFT-SLE. We also modified the implementation of the Link-Free algorithm. While the original Link-Free algorithm does not explicitly persist data structure links, it still allocates the links from persistent memory. We can achieve better performance by allocating the links from volatile memory. To emphasize the difference we refer to this as LF-V2.

3.1 Our Persistent List Implementations

In order to compare our extension to existing work we provide several implementations of persistent linked-lists which utilize our extended-link-and-persist approach. We choose to implement and study linked lists because they generally do not require complicated volatile synchronization.

```
def PersistenceFreeContains(key) :
    p = head, pNext = p.next, curr = UnmarkPtr(pNext)
    while true :
        if curr.key ≤ key : break
        p = curr, pNext = p.next
        curr = UnmarkPtr(pNext)
    hasKey = curr.key==key
    if IsDurable(ptNext) : return hasKey
    old1 = p.old, pNext2 = p.next, old2 = p.old
    pDiff = pNext≠pNext2, oldDiff = old1≠old2
    if pDiff or oldDiff or old1==null : return hasKey
    if IsIFlagged(old1) : return false
    if hasKey : return true
    return UnmarkPtr(old1).key==key
```

Algorithm 1: Pseudocode for the persistence-free contains function of our Physical-Delete (PD) list. The volatile synchronization is based on the list of Fomitchev and Ruppert.

We refer to our implementations as PD (Physical-Delete), PD-S (SWCAS implementation of PD), LD (Logical-Delete) and LD-S (SWCAS implementation of LD). The names refer to the synchronization approach and primitive. Our implementations use two different methods for achieving synchronization in volatile memory. Specifically we use one based on the Harris list [13] and another based on the work of Fomitchev and Ruppert [10]. The former takes a lazy approach to deletion that relies on marking for logical deletion and helping. As a result, marked pointers must be written to persistent memory which requires an extra psync. The latter does not take a lazy approach to deletions but still relies on helping and requires extra volatile memory synchronization through the use of marking and flagging. Fortunately, we do not need to persist marked or flagged pointers with this approach. Figure 1 shows an example of an update operation in the PD list implementation. We also implement separate variants using 2 different synchronization primitives, DWCAS and SWCAS. Table 1 summarizes some of the details of these approaches. We assume that the size of the *key* and *value* fields allow a single node to fit on one cache line meaning a *flush* on any field of the node guarantees that all fields are written to persistent memory. The assumption that the data we want to persist fits on a single cache line is common. David et al., Zuriel et al. and several others have relied on similar assumptions [21, 9, 8, 18]. It is possible that our persistent list could be modified to allow for the case where nodes do not fit onto a single cache line by adopting a strategy similar to [6].

Search Variants. As part of our persistent list, we implement 4 variants of the `contains` operation: persist all, asynchronous persist all, persist last and persistence free. We focus on the latter two since the others are naive approaches that perform many redundant psyncs.

Persist Last (PL). If the pointer into the terminal node of the traversal is marked as not persistent then write it to persistent memory and set its persis-

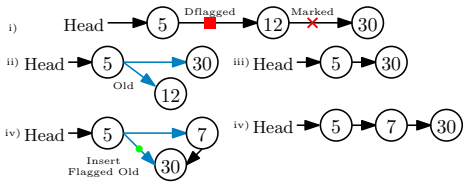


Fig. 1: Steps to execute an `insert(7)` operation in our PD list implementation. Blue pointers indicate non-durable pointers (with persistence bits set to 0). i) Initially we have three nodes. The node containing 5 has a pending delete flag (Dflagged) and the node containing 12 is marked for deletion. We traverse to find a key ≥ 7. ii) Help finish the pending `Remove` via DWCAS to unlink marked node and set old pointer. iii) Flush and set persistence bit via DWCAS (clearing old pointer). iv) Via DWCAS insert 7 and set old pointer. The old pointer is flagged to indicate a pending insert. v) Flush and set persistence bit via DWCAS.

tence bit via a CAS. This variant is the most similar to the searches in the linked list implemented using the original Link-and-Persist technique.

Persistence Free (PF). If the pointer into the terminal node of the traversal performed by the search is marked as not persistent then use the information in the *old field* of the node's predecessor to determine the correct return value without performing any persistence events. Since we do not need to set the durability bit of any link, this variant does not perform any writes and never performs a psync. Algorithm 1 shows the pseudocode for the persistence-free search of the PD list. For simplicity we abbreviate some of the bitwise operations with named functions. Specifically, *UnmarkPtr* which removes any marks or flags, *IsDurable* which checks if the pointer is marked as persistent and *IsIflagged* which checks if the pointer was flagged by an `insert`.

Theorem 3. *The PD, PD-S, LD, and LD-S lists are durable linearizable and lock-free.*

We prove Theorem 3 in the full version of the paper. We can also show that our list implementations are durable linearizable by considering a volatile abstract set (the keys in the list that are reachable in volatile memory) and a persistent abstract set (the keys in the list that are reachable in persistent memory). By identifying, for each operation, the points at which these sets change, we can show that updates change the volatile abstract set prior to changing the persistent abstract set and that each update changes the the volatile abstract set exactly once. It follows that the list is always consistent with some persistent abstract set.

If we never invoke a persistence-free `contains` operation then we can prove that the implementations are SLE linearizable and lock-free. Doing so simply requires that we change our arguments regarding when we linearize update operations such that the linearization point is not after the CPE. Note that of the

Name	Synch. Approach	Synch. Primitive	Min Psyncs Per Insert/Remove	
PD	Fomitchev	DWCAS	1	1
PD-S	Fomitchev	SWCAS	2	1
LD	Harris	DWCAS	1	2
LD-S	Harris	SWCAS	2	2

Table 1: Our Novel Persistent List Details.

set implementations that we discuss, those that have persistence-free searches are examples of implementations which are strict linearizable but not SLE linearizable. These implementations require that the recovery procedure or operations invoked after a crash take steps which effectively linearize operations. This is because following a crash, one cannot tell the difference between an operation that has progressed far enough to allow some future operation to help linearize it and an operation that was already linearized.

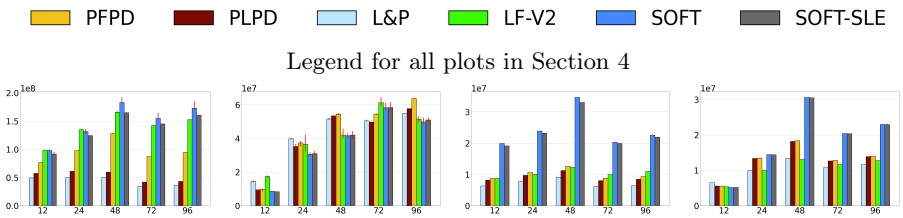
4 Evaluation

We present an experimental analysis of our persistent list compared to existing persistent lists on various workloads. We test our variants of the `contains` operation separately meaning no run includes more than one of the variants.[4] To distinguish between our implementations of the `contains` operation we prefix the names of our persistent list algorithms with the abbreviation of a `contains` variant (for example PFLD refers to one of our persistent lists which utilized only Persistence-Free searches and the Logical-Deletion algorithm). Due to space constraints we only present the best performing implementations of our persistent list. We test the performance of these lists in terms of throughput (operations per second). We also examine the psync behaviour of these algorithms. Specifically, we track the number of psyncs that are performed by searches and the number of psyncs that are performed by update operations.

All of the experiments were run on a machine with 48 cores across 2 Intel Xeon Gold 5220R 2.20GHz processors which provides 96 available threads (2 threads per core and 24 cores per socket). The system has a 36608K L3 cache, 1024K L2 cache, 64K L1 cache and 1.5TB of NVRAM. The NVRAM modules installed on the system are Intel Optane DCPMMs. We utilize the same benchmark as [3] for conducting the empirical tests. Keys are accessed according to a uniform distribution. We prefill the lists to 50% capacity before collecting measurements. Each test consisted of ten iterations where each individual test ran for ten seconds. The graphs show the average of all iterations. Libvmmalloc was the persistent memory allocator used for all algorithms.

Throughput. Figure 2 shows the throughput of our best persistent list variants compared to the existing algorithms. Since the DWCAS implementation of our list out performed the SWCAS implementation we compare only our DWCAS implementations. SOFT performs best when there is high contention in read dominant workloads and consistently best for non-read dominant workloads.

[4] Source code: https://gitlab.com/Coccimiglio/setbench



(a) 99% Search K=50 (b) 99% Search K=500 (c) 50% Search K=50 (d) 50% Search, K=500

Fig. 2: Persistent list throughput. X-axis: number of concurrent threads. Y-axis: operations/second. K is the list size.

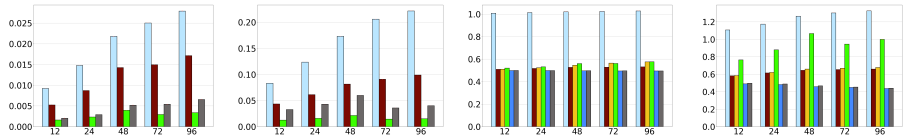
Lesson learned: Persisting more information in update operations is generally more costly but persistence free searches do not seem to provide major performance improvements.

Psync Behaviour. The recent trend to persist less data structure state has influenced implementations of persistent objects focused on minimizing the amount of psyncs required per operation. We know that SLE linearizable algorithms cannot have persistence-free searches. From [7] we also know that update operations require at least 1 psync. Of the persistent lists that we consider, the persistent lists from [21] are unique in that the the maximum number of psyncs per update operation is bounded. To better understand the cost incurred by psyncs, we track the number of psyncs performed by read-only operations (searches) and the number of psyncs performed by update operations. Note that for updates this includes unsuccessful updates which might not need to perform a psync. Figure 3 shows the average number of psyncs per search and the average number of psyncs per update operation. We observe that searches rarely perform a psync in any of the algorithms that do not have persistence-free searches. On average, update operations do not perform more than the minimum number of required psyncs.

Lesson learned: Algorithmic techniques such as *persistence bits* for reducing the number of psyncs are highly effective. On average, there are very few redundant psyncs in practice.

Recovery. It is not practical to force real system crashes in order to test the recovery procedure of any algorithm. It is possible that one could simulate a system crash by running the recovery procedure as a standalone algorithm on an artificially created memory configuration. This is problematic because the recovery procedure of a durable linearizable algorithm is often tightly coupled to some specific memory allocator (this is true of the existing algorithms that we consider). This makes a fair experimental analysis of the recovery procedure difficult. It is easier to describe the worst case scenario for recovering the data structure for each of the algorithms. To be specific, we describe the worst case persistent memory layout produced by the algorithm noting how this relates to the performance of the recovery procedure.

The Link-Free list does not persist data structure links. As a result, there is no way to efficiently discover all valid nodes meaning the recovery procedure might



(a) 99% Search (b) 50% Search (c) 99% Search (d) 50% Search

Fig. 3: Psync Behaviour. X-axis: number of concurrent threads. (a), (b) Y-axis: average psyncs/search, (c), (d) Y-axis: average psyncs/update. List size is 50.

need require traversing all of the memory. The allocator utilized by Zuriel et al partitions memory into *chunks*. We can construct a worse case memory layout for the recovery procedure as follows: Suppose that we completely fill persistent memory by inserting keys into the list. Now remove nodes such that each chunk of memory contains only one node at an unknown offset from the start of the chunk. To discover all of the valid nodes the recovery procedure must traverse the entire memory space. The SOFT list also does not persist data structure links. The requirements of the recovery procedure for SOFT list is the same as the Link-Free list. We can construct the worst case memory layout for the recovery procedure in the same way as we did for the Link-Free list yielding the same asymptotic time complexity. The Link-and-Persist list can utilize an empty recovery procedure. The actual recovery procedure for the list implemented by the authors of [9] does extra work related to memory reclamation.

We utilize DWCAS and asynchronous flush instructions to achieve a minimum of one psync per `insert` operation. There are some subtleties with this implementation that result in a recovery complexity which is $O(N+n)$ for a list containing N nodes and a maximum of n concurrent processes. Implementations that use SWCAS (or DWCAS allowing for a minimum of two psyncs per `insert`) can utilize an empty recovery procedure.

Lesson learned: If structure is persisted, recovery can be highly efficient. Without any persisted structure, recovery must traverse large regions (or even all) of shared memory.

SLE linearizable vs. Durable linearizable Sets. We have seen that there exists a theoretical separation between SLE linearizable and durable linearizable objects. For persistent lists we observe that this separation does not lead to significant performance differences in practice. 4 of the algorithms (Figure 2) are SLE linearizable. Specifically, our PLPD list, the L&P list, LF list, and SOFT-SLE. The SOFT list and our PFPD list which both use persistence-free searches are durable linearizable. The high cost of a psync and the impossibility of persistence-free searches in a SLE linearizable lock-free algorithm would suggest that the SLE linearizable algorithms that we test should perform noticeably worse. In practice, it is true that for most of the tested workloads, the algorithms that have persistence-free searches perform best (primarily SOFT). However, for many workloads, performance of SLE linearizable algorithms are comparable to the durable linearizable algorithms. In fact, for some workloads, the SLE linearizable lists perform better than the durable linearizable alternatives.

Lesson learned: SLE linearizable algorithms can be fast in practice, despite theoretical tradeoffs.

5 Discussion

We prove that update operations in durable linearizable lock-free sets will perform at least one redundant psync. We motivate the importance of ensuring limited effect for sets and defined strict limited effect (SLE) linearizability for sets. We prove that SLE linearizable lock-free sets cannot have persistence-free reads. We implement several persistent lists and evaluate them rigorously. Our experiments demonstrate that SLE linearizable lock-free sets can achieve comparable or better performance to durable linearizable lock-free sets despite the theoretical separation. For the algorithms and techniques that we examined, supporting persistence-free reads is what separates the durable linearizable sets from the SLE linearizable. However, the SLE linearizable sets rarely perform a psync during a read. For those researchers that value ensuring limited effect for sets but are unsure about the performance implications, we recommend beginning with SLE linearizable implementations since a SLE linearizable implementation may not have much overhead and it may be sufficient for the application. Our work also exposes that psync complexity is not a good predictor of performance in practice, thus motivating need for better metrics to compare persistent objects.

In this work we focused specifically on sets because we wanted to understand the psync complexity of a relatively simple data structure like sets. We think that there is clear potential to generalize our theoretical results to other object types or classes of object types and perform similar empirical analysis of persistent algorithms for those objects, thus bridging the gap between theory and practice.

Acknowledgements This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Launch Supplement: DGECR-2019-00048, NSERC Discovery Program grant: RGPIN-2019-04227, and the University of Waterloo.

References

1. Aguilera, M.K., Frolund, S.: Strict linearizability and the power of aborting. Tech. rep., HP Laboratories Palo Alto (2003)
2. Berryhill, R., Golab, W.M., Tripunitara, M.: Robust shared objects for non-volatile main memory. In: 19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France. pp. 20:1–20:17 (2015)
3. Brown, T., Prokopec, A., Alistarh, D.: Non-blocking interpolation search trees with doubly-logarithmic running time. In: Proceedings of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. pp. 276–291 (2020)
4. Censor-Hillel, K., Petrank, E., Timnat, S.: Help! In: Proceedings of the 2015 ACM Symp. on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015. pp. 241–250 (2015)

5. Coccimiglio, G., Brown, T., Ravi, S.: The fence complexity of persistent sets (2023), https://mc.uwaterloo.ca/pubs/fence_complexity/fullpaper.pdf, full version of this paper
6. Cohen, N., Friedman, M., Larus, J.R.: Efficient logging in non-volatile memory by exploiting coherency protocols. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–24 (2017)
7. Cohen, N., Guerraoui, R., Zablotchi, I.: The inherent cost of remembering consistently. In: Proceedings of the 30th on Symp. on Parallelism in Algorithms and Architectures. pp. 259–269 (2018)
8. Correia, A., Felber, P., Ramalhete, P.: Persistent memory and the rise of universal constructions. In: Proceedings of the Fifteenth European Conference on Computer Systems. pp. 1–15 (2020)
9. David, T., Dragojevic, A., Guerraoui, R., Zablotchi, I.: Log-free concurrent data structures. In: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). pp. 373–386 (2018)
10. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the twenty-third annual ACM Symp. on Principles of distributed computing. pp. 50–59 (2004)
11. Friedman, M., Ben-David, N., Wei, Y., Blelloch, G.E., Petrank, E.: Nvtraverse: in nvram data structures, the destination is more important than the journey. In: Proceedings of the 41st ACM SIGPLAN Conf. on Programming Language Design and Impl. pp. 377–392 (2020)
12. Friedman, M., Petrank, E., Ramalhete, P.: Mirror: making lock-free data structures persistent. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1218–1232 (2021)
13. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. pp. 300–314 (2001)
14. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
15. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: International Symp. on Distributed Computing. pp. 313–327. Springer (2016)
16. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., Zhao, J., Swanson, S.: Basic performance measurements of the intel optane dc persistent memory module. arXiv preprint arXiv:1903.05714 (2019)
17. Peng, I.B., Gokhale, M.B., Green, E.W.: System evaluation of the intel optane byte-addressable nvm. In: Proceedings of the International Symp. on Memory Systems. pp. 304–315 (2019)
18. Ramalhete, P., Correia, A., Felber, P.: Efficient algorithms for persistent transactional memory. In: Proceedings of the 26th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. pp. 1–15 (2021)
19. Wang, T., Levandoski, J., Larson, P.A.: Easy lock-free indexing in non-volatile memory. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). pp. 461–472. IEEE (2018)
20. Wei, Y., Ben-David, N., Friedman, M., Blelloch, G.E., Petrank, E.: Flit: A library for simple and efficient persistent algorithms. arXiv preprint arXiv:2108.04202 (2021)
21. Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., Petrank, E.: Efficient lock-free durable sets. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–26 (2019)