Efficient Lock-Free Durable Sets*

YOAV ZURIEL, Technion, Israel
MICHAL FRIEDMAN, Technion, Israel
GALI SHEFFI, Technion, Israel
NACHSHON COHEN†, Amazon, Israel
EREZ PETRANK, Technion, Israel

Non-volatile memory is expected to co-exist or replace DRAM in upcoming architectures. Durable concurrent data structures for non-volatile memories are essential building blocks for constructing adequate software for use with these architectures. In this paper, we propose a new approach for durable concurrent sets and use this approach to build the most efficient durable hash tables available today. Evaluation shows a performance improvement factor of up to 3.3x over existing technology.

CCS Concepts: • **Hardware → Non-volatile memory**; • **Software and its engineering → Concurrent programming structures**.

Additional Key Words and Phrases: Concurrent Data Structures, Non-Volatile Memory, Lock Freedom, Hash Maps, Durable Linearizability, Durable Sets

ACM Reference Format:
Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 128 (October 2019), 26 pages. https://doi.org/10.1145/3360554

1 INTRODUCTION

An up-and-coming innovative technological advancement is non-volatile RAM (NVRAM). This new memory architecture combines the advantages of DRAM and SSD. The latencies of NVRAM are expected to come close to DRAM, and it can be accessed at the byte level using standard `store` and `load` operations, in contrast to SSD, which is much slower and can be accessed only at a block level. Unlike DRAM, the storage of NVRAM is persistent, meaning that after a power failure and a reset, all data written to the NVRAM is saved [Zhang and Swanson 2015]. That data, in turn, can be used to reconstruct a state similar to the one before the crash, allowing continued computation.

Nevertheless, it is expected that caches and registers will remain volatile [Izraelevitz et al. 2016]. Therefore, the state of data structures underlying standard algorithms might not be complete in the NVRAM view, and after a crash this view might not be consistent because of missed writes that were in the caches but did not reach the memory. Moreover, for better performance, the processor

*This work was supported by the Israel Science Foundation grant No. 274/14
†Work done in cooperation with the Technion (external and not related to the author's work at Amazon, but done during employment time).

Authors' addresses: Yoav Zuriel, CS Department, Technion, Israel, yoavzuriel@cs.technion.ac.il; Michal Friedman, CS Department, Technion, Israel, michal.f@cs.technion.ac.il; Gali Sheffi, CS Department, Technion, Israel, galish@cs.technion.ac.il; Nachshon Cohen, Amazon, Israel, nachshonc@gmail.com; Erez Petrank, CS Department, Technion, Israel, erez@cs.technion.ac.il.

2475-1421/2019/10-ART128
https://doi.org/10.1145/3360554

may change the order in which writes reach the NVRAM, making it difficult for the NVRAM to even reflect a consistent prefix of the computation. In simpler words, the order in which values are written to the memory may be different from the program order. Thus, the implementations and the correctness conditions for programs become more involved.

Harnessing durable storage requires the development of new algorithms that can ensure a consistent state of the program in memory when a crash occurs and the development of corresponding recovery mechanisms. These algorithms need to write back cache lines explicitly to the NVRAM, to ensure that important stores persist in an adequate order. The latter can be obtained using a FLUSH instruction that explicitly writes back cache lines to the DRAM. Flushes typically need to be accompanied by a memory fence in order to guarantee that the write back is executed before continuing the execution. This combination of instructions is denoted psync. The cost of flushes and memory fences is high, hence their use should be minimized to improve performance.

When dealing with concurrent data structures, *linearizability* is often used as the correctness definition [Herlihy and Wing 1990]. An execution is *linearizable* if every operation seems to take effect instantaneously at a point between its invocation and response. Various definitions of correctness for durable algorithms have been proposed. These definitions extend linearizability to the setting that includes crashes, recoveries, and flush events. In this work, we adopt the definition of Izraelevitz et al. [2016] denoted *durable linearizability*. Executions in this case also include crashes alongside invocations and responses of operations. Intuitively, an execution is *durable linearizable* if all operations that survive the crashes are linearizable.

This work is about implementing efficient set data structures for NVRAM. Sets (most notably hash maps) are widely used, e.g., for key-value storage [Debnath et al. 2010; Nishtala et al. 2013; Raju et al. 2017]. It is, therefore, expected that durable sets would be of high importance when NVRAMs reach mass production. The durable sets proposed in this paper are the most efficient available today and can yield better throughput for systems that require fault-tolerance. Our proposed data structures are all lock-free, which make them particularly adequate for the setting. First, lock-free data structures are naturally efficient and scalable [Herlihy and Shavit 2008]. Second, the use of locks in the face of crashes requires costly logging to undo instructions executed in a critical section that did not complete before the crash. Nesting of locks may complicate this task substantially [Chakrabarti et al. 2014].

State-of-the-art constructions of durable lock-free sets, denoted *Log-Free Data Structures*, were recently presented by David et al. [2018]. They proposed two clever techniques to optimize durable structures and built four implementations of sets. Their techniques were aimed at reducing the number of required explicit write backs (psync operations) to the non-volatile memory.

In this paper, we present a new idea with two algorithms for durable lock-free sets, which reduce the required flushes substantially. Whereas previous work attempted to reduce flushes that were not absolutely necessary for recovery, we propose to completely avoid persisting any pointer in the data structure. In a crash-free execution, we can use the pointers to access data quickly, but when a crash occurs, we do not need to access a specific key fast. We only need a way to find all nodes to be able to decide which belong to the set and which do not. This idea is applicable to a set because for a set we only care if a node (which represents a key) belongs to the data structure or not. Thus, we only persist the nodes that represent set members by flushing their content to the NVRAM, but we do not worry about persisting pointers that link these nodes -- hence the name *link-free*. The persistent information on the nodes allows determining (after a crash) whether a node belongs to the set or not. We also allow access to all potential data structure nodes after a crash so that during recovery we can find all the members of the set and reconstruct the set data structure. We do that by keeping all potential set nodes in special designated areas, which are accessible after a crash.

In volatile memory, we still use the original pointers of the data structure to allow fast access to the set nodes, e.g., by keeping a hash map (in the volatile memory) that allows fast access to members of the set. Not persisting pointers significantly reduces the number of flushes (and associated fences), thereby, drastically improving the performance of the obtained durable data structure. To recover from a crash, the recovery algorithm traverses all potential set nodes to determine which belong to the set. The recovery procedure reconstructs the full set data structure in the volatile space, enabling further efficient computation.

The first algorithm that we propose, called *link-free*, implements the idea outlined in the above discussion in a straightforward manner. The second algorithm, called SOFT, attempts to further reduce the number of fences to the minimum theoretical bound. This achievement comes at the expense of algorithmic complication. Without flushes, the first (link-free) algorithm would probably be more performant, as it executes fewer instructions. Nevertheless, in the presence of flushes and fences, the second (SOFT) algorithm often outperforms link-free. Interestingly, SOFT executes at most one fence per thread per update operation. It has been shown in [Cohen et al. 2018] that there are no durable data structures that can execute fewer fences in the worst case. Thus, SOFT matches the theoretical lower bound, and is also efficient in practice.

On top of the innovative proposal to avoid persisting pointers (and its involved implementation), we also adopt many clever techniques from previous work. Among them, we employ the link-and-persist technique from David et al. [2018] that uses a flag to signify that an address has already been flushed so that further redundant psync operations can be avoided. Another innovative technique follows an observation in Cohen et al. [2017] that flushes can be elided when writing several times to the same cache line. In such case, it is sufficient to use fences (or, on a TSO platform, only compiler fences) to ensure the order of writes to cache and the same order is guaranteed also when writing to the NVRAM. Each write back of this cache line to the memory always reflects a prefix of the writes as executed on the cache line.

Both schemes are applicable to linked lists, hash tables, skip lists and binary search trees and both guarantee lock-freedom and maintain a consistent state upon a failure. We implemented a basic durable lock-free linked list and a durable lock-free hash table based on these two schemes and evaluated them against the durable lock-free linked list and hash map of David et al. [2018]. The code for these implementations is publicly available in GitHub at https://github.com/yoavz1997/Efficient-Lock-Free-Durable-Sets. Our algorithms outperform previous state-of-the-art durable hash maps by a factor of up to 3.3x.

The basic assumption in this work (as well as previous work mentioned) is that crashes are infrequent, as is the case for servers, desktops, laptops, smartphones, etc. Therefore, efficiency is due to low overhead on data structures operation. The algorithms proposed here do not fit a scenario where crashes are frequent. Substantial work on dealing with scenarios in which crashes are frequent has been done. The research focuses on energy harvesting devices in which power failures are an integral part of the execution, e.g., [Colin and Lucia 2016; Jayakumar et al. 2015; Lucia et al. 2017; Maeng et al. 2017; Maeng and Lucia 2018; Ruppel and Lucia 2019; Woude and Hicks 2016; Yıldırım et al. 2018]. Some of these devices also have a non-volatile memory (FRAM) and volatile registers. To deal with the frequent crashes, programs are executed by using checkpoints (enforced by the programmer, by the compiler, by run time, or by special hardware), and thus achieve persistent execution. Currently, those approaches do not deal with concurrency or with durable linearizability.

The rest of the paper is organized as follows. In Section 2 we provide an overview of the set algorithms. Sections 3 and 4 provide the details of the *link-free* and *SOFT* algorithms, respectively. In Section 5 we discuss the memory management scheme used. The evaluation is laid out in Section 6. Finally, we examine related work in Section 7, and conclude in Section 8. Formal correctness proofs

for the link-free list and the SOFT list are laid out in Appendices B and C correspondingly, in the full version of the paper (can be found in at https://arxiv.org/abs/1909.02852).

2 OVERVIEW OF THE PROPOSED DATA STRUCTURES

A *set* is an abstract data structure that maintains a collection of unique keys. It supports three basic operations: *insert*, *remove*, and *contains*. The *insert* operation adds a key to the set if the key is not already in the set. The *remove* operation deletes the given key from the set (if the key belongs to the set) and the *contains* operation checks whether a given key is in the set. A key in a set is usually associated with some data. In our implementation we assume this data comprise one word. Our scheme can easily be extended to support other forms of data or no data at all.

A typical implementation of a lock-free set relies on a lock-free linked graph, such as a linked list, a skip list, a hash table, or a binary search tree (e.g., [Harris 2001; Herlihy and Shavit 2008; Michael 2002; Natarajan and Mittal 2014; Shalev and Shavit 2006]). Each node typically represents a single key and consists of a key, a value, and a *next* pointer(s) to one (or more) additional nodes in the set. The structure of the linking pointers determines the set complexity, from a simple linked list (i.e., a single *next* pointer) to skip lists or binary search trees.

One way to transform a lock-free set into a durable one[1] is to ensure that the entire structure is kept consistent in the NVRAM [Izraelevitz et al. 2016]. Using this method, each modification to the set has to be written immediately to the NVRAM. When reading from the set, readers are also required to flush the read content, to avoid acting according to values that would not survive a crash. Upon recovery, the content of the data structure in the non-volatile memory matches a consistent prefix of the execution. The problem with this approach is that the large number of flushes imposes a high performance overhead.

In this paper, we take a different approach that fits data structures that represent sets. Instead of keeping the entire structure in NVRAM, we only ensure that the key and the value of each node are stored durably. In addition, we maintain a persistent state in each node, which lets the recovery procedure determine whether the insertion of a specific node has been completed and whether this node has not been removed. By providing such per-node information, we avoid needing to keep the linking structure (i.e., *next* pointers) of the set.

Both of our set algorithms maintain a basic unit called the *persistent node*, consisting of a key, a value and a Boolean method for determining whether the key in the node is a valid member of the set. The persistent nodes are allocated in special *durable areas*, which only contain persistent nodes. During execution, the system manages a collection of durable areas from which persistent nodes are allocated. Following a crash, the recovery procedure iterates over the durable areas and reconstructs the data structure with all its volatile links from all valid nodes.

A major challenge we face in the design of our algorithms is to ensure that the order in which operations take effect in the non-volatile memory view matches some linearization order of the operations executed in the volatile memory. This match is required to guarantee the durable linearizability of the algorithms.

One standard techniques employed in the proposed algorithms is the marking of nodes as *removed* by setting the least significant bit of one of the node's pointers. This method was presented by Harris [2001] and was used in many subsequent algorithms. The algorithms we propose extend lock-free algorithms that employ this method. In the description, we say "mark a node" to mean that a node is marked for removal in this manner.

[1] When saying an algorithm is durable we mean the algorithm is durable linearizable [Izraelevitz et al. 2016].

2.1 Recovery

The recovery procedure traverses all areas that contain persistent nodes. It determines the nodes that currently belong to the set and reconstructs the linked data structure in the volatile memory to allow subsequent fast access to the nodes. Note that this construction does not need to use psync operations. Moreover, the reconstructed set may have a different structure from the one prior to the crash (for example, as a randomized skip list). The sole purpose of the structure is to make normal operations efficient.

The proposed algorithms require the recovery execution to complete before further operations can be applied. Before completing recovery of the data structure on the volatile memory, the data structure is not coherent and cannot be used. This is unlike some previous algorithms, such as [David et al. 2018; Friedman et al. 2018], which allow the recovery and subsequent operations to run concurrently. This requirement works well in a natural setting where crashes are infrequent.

2.2 Link-Free Sets

The first algorithm we propose for implementing a durable lock-free set is called *link-free*, as it does not persist links. This algorithm keeps two validity bits in each node, allowing making a node as invalid while it is in a transient state before being inserted into the list. A node is considered valid only if the value of both bits match. Deciding if a node is in the set depends on whether it is valid and not logically deleted. We follow Harris [2001] and mark a node to make it logically deleted. The complementary case is when the validity bits do not match, making the node invalid. An invalid node is not in the set.

To determine whether a node is in the set or not, the *contains* operation checks that it is in the volatile set structure, i.e., that it is not marked as deleted. If this is the case, the contains operation makes sure this node is valid and flushed so that this node will be resurrected if a crash and a recovery occur. This ensures that the returned value of the contains matches the NVRAM view of the data structure's state.

To insert a node, the node first needs to be initialized. To this end, one validity bit is flipped, making the node invalid, and then the key and value are written into it. Intermediate states do not affect a future recovery because an invalid node is not recovered. Afterwards, the node is inserted into the linked structure and is made valid by flipping the second validity bit. The insertion completes by executing a psync on the new node, making the node durably in the set. If a node with the same key already exists, the previous insert is first helped by making the previously inserted node valid, and ensuring its content is flushed. At this point, the *insert* can return and report failure due to the key already existing in the set.

To remove a node, the removal first helps complete the insertion of the target node. The node is made valid and then its *next* pointer can be marked, so that it becomes logically deleted. The removal is completed by executing a psync on the marked node. If the node is already logically deleted, it is flushed using a psync and the thread returns reporting failure (as it was already deleted). During recovery, a marked node is considered not in the set.

Note that psync may be called multiple times on the same node. To further reduce the number of psync operations, we employ an optimization. Since the proposed algorithm persists a newly inserted node and a newly marked one, we use two flags to indicate whether a psync was executed after inserting the node or after deleting it. The first flag indicates that a new node was written to the NVRAM, and the second flag indicates that a deleted node was written back. Before actually calling psync on the node, the insert (or remove, correspondingly) flag is checked to minimize the number of redundant psync operations. After calling psync on a node, the insert (or remove, correspondingly) flag is set. This way threads coming in a later point see that the flags are set, and

they do not execute an unnecessary psync. This is an extension of the *link-and-persist* optimization presented by David et al. [2018].

2.3 SOFT: Sets with an Optimal Flushing Technique

The second algorithm we introduce is *SOFT* (Sets with an Optimal Flushing Technique). SOFT is also a durable lock-free algorithm for a set. It requires the minimal theoretical number of fences per operation. Specifically, each thread performs at most one fence per update and zero fences per read operation [Cohen et al. 2018].

Each key in the set has two separate representations in memory: the persistent node and the volatile node. Similarly to our link-free algorithm, *persistent nodes* (PNodes) are stored in the durable areas. They contain a key and its associated value and three validity bits used for a similar but extended validity scheme. Each time we wish to write to the NVRAM, we do so via a PNode method. The PNode methods are described in further detail in Section 4.1.

The volatile node takes part in the volatile-linked graph of the set. In addition to holding the key and value, it has a pointer to a PNode with the same key and value, and pointers to its descendants in the linked structure. The pointer, which is usually used for marking, is used to keep a state that indicates the condition the node is in. A node can be in one of the following four states:

(1) Inserted: The node is in the set, is linked to the structure in the volatile memory and its PNode has been written to the NVRAM.
(2) Deleted: The node is not in the set. In this case, the node can be unlinked from the volatile structure and later freed.
(3) Intention to Insert: The node is in the middle of being inserted, and its PNode is not yet guaranteed to be written to the NVRAM.
(4) Inserted with Intention to Delete: The node is in the middle of being removed, and its removed condition is not yet guaranteed to be written to the NVRAM.

The read operation (contains) executes on the volatile structure and does not require any psync operations, which is in line with the bound. A contains operation only reads the state of the relevant node and acts accordingly. A node that is either "inserted" or "inserted with intention to delete" is considered a part of the set, so contains returns `true`. Nodes with one of the remaining states ("intention to insert" or "deleted") cause the contains operation to return `false`.

To add a node to the set, SOFT allocates a volatile node and a PNode, links them together, and fixes the volatile node's state to be "intention to insert". Next, the insert operation adds the node to the volatile structure. Read operations seeing the node in this state do not consider it as a part of the set. Thereafter, the associated PNode is written to the NVRAM and the state of the volatile node is changed to "inserted". When the state is "inserted", other operations see the key of this node as a part of the set.

When trying to insert a node into the volatile structure, if there is a node with the same key in the set, the node's state is checked. If the state of this node is "inserted" or "inserted with intention to delete", the node might be in the set in the event of a crash, so the thread fails right away. If the state is "intention to insert", then the old node is not yet in the set, so the current thread helps complete the insertion before failing. Just as many other algorithms, in SOFT, deleted nodes are trimmed when traversing the linked-structure of the set, so there is no need to consider the scenario of seeing a node with the "deleted" state. Either way, only a single psync operation is executed, following the theoretical bound.

When a remove operation wishes to remove a node, it must ensure the relevant node is in the set. A remove operation changes the node's state from "inserted" to "inserted with intention to delete". In this case, read operations do acknowledge the node because the removal has not finished yet.

Then the removal is written to the NVRAM and, finally, the state changes to "deleted". A node with the state "intention to insert" cannot be removed because it is not yet in the set. In this case, the remove operation can return a failure: there is no node in the set with the given key. Alternatively, the state of the node the thread wishes to remove may already be "inserted with intention to delete". In this case, before failing, the thread helps completing the removal and persisting it. Just as before, this operation is done using only a single psync.

The goal of the states is to make threads help each other complete operations and reduce the number of psync operations to the minimum. States 3 and 4, described above, are used as flags to indicate the beginning of an operation so other threads are able to help.

Both insert and remove use the same logic. They first update the non-volatile memory, and only then execute the operation (reaching a linearization point) on the volatile structure. In other words, the state a thread sees in SOFT already resides in the NVRAM, unlike link-free in which a node has to be written back to the NVRAM. This logic follows the upper bound of Cohen et al. [2018].

3 THE DETAILS OF THE LINK-FREE ALGORITHM

In this section we described the *link-free* linked list. A link-free hash table is constructed simply as a table of buckets, where each bucket uses the link-free list to hold its items. Extending this algorithm to a skip list is straightforward.

The link-free linked list uses a node to store an item in the set; see Listing 1. Unlike SOFT, each key has a single representation in both volatile and non-volatile space. Each node has two validity bits, two flags to reduce the number of psync operations, a key, a value and a *next* pointer that also contains a marking bit to indicate a logical deletion [Harris 2001].

Listing 1. Node Structure

```
1 class Node{
2     atomic<byte> validityBits;
3     atomic<bool> insertFlushFlag, deleteFlushFlag;
4     long key;
5     long value;
6     atomic<Node*> next;
7 } aligned(cache line size);
```

Building on the implementation of Harris [2001], the list is initialized with a head sentinel node with key $-\infty$, and a tail sentinel node with key ∞. All the other nodes are inserted between these two, and are sorted in an ascending order.

3.1 Auxiliary Functions

Before explaining each operation, we first discuss the auxiliary functions. We use the functions isMarked, getRef, and mark without providing their implementations since these are only bit operations, to clean, mark, or test the least significant bit of a pointer. In addition, we use FLUSH_DELETE and FLUSH_INSERT to execute a psync operation to write the content of a node to the NVRAM when removing or inserting it from or into the list. Before executing the psync, the appropriate (insert or delete) flag is used to check whether the latest modification to this node has already been written to the NVRAM so avoid repeated flushing. Next, flipV1 and makeValid modify the validity of a node: flipV1 flips the value of the first validity bit, making the node invalid, and makeValid makes the node valid by equating the value of the second bit to the value of the first bit.

The auxiliary function trim (Listing 2) unlinks curr from the list. Just prior to the unlinking CAS (line 4), node curr is flushed to make the delete mark on it persistent (line 2). The return value signifies whether the unlinking succeeded or not.

The `find` function (Listing 2) traverses the list in order to locate nodes `curr` and `pred`. The key of `curr` is greater or equal to the given key, and `pred` is the predecessor of `curr` in the list. During its search of the list, `find` invokes `trim` on any marked (logically deleted) node (line 16).

Listing 2. Auxiliary functions

```
1  bool trim(Node *pred, Node *curr){
2     FLUSH_DELETE(curr);
3     Node *succ = getRef(curr->next.load());
4     return pred->next.compare_exchange_strong(curr, succ);
5  }
6
7  Node*, Node* find(long key){//method returns two pointers, pred and curr.
8      Node* pred = head, *curr = head->next.load();
9      while(true){
10         if(!isMarked(curr->next.load())){
11             if(curr->key >= key)
12                 break;
13             pred = curr;
14         }
15         else
16             trim(pred, curr);
17         curr = getRef(curr->next.load());
18     }
19     return pred, curr;
20 }
```

3.2 The contains Operation

The contains operation, based on the optimization of Heller et al. [2006], is wait-free unlike the lock-free insert and remove operations. Given a key, it returns `true` if a node with that key is in the list and `false` otherwise.

In lines 3 – 4 (Listing 3), the list is traversed in order to find the requested key. If a node with the given key is not found, then the operation returns `false` (line 5). If the node exists but has been marked, it is flushed and the thread returns `false` (line 7). The last possible case is that the node exists and has not been marked as removed. In this case, the node is made valid, is flushed to make its insertion visible after a crash, and `true` is returned (line 11).

Listing 3. Link-Free List contains

```
1  bool contains(long key){
2      Node* curr = head->next.load();
3      while(curr->key < key)
4          curr = getRef(curr->next.load());
5      if(curr->key != key)
6          return false;
7      if(isMarked(curr->next.load())){
8          FLUSH_DELETE(curr);
9          return false;
10     }
11     makeValid(curr);
12     FLUSH_INSERT(curr);
13     return true;
```

```
14 }
```

3.3 The insert Operation

The insert operation adds a key-value pair to the list. It returns `true` if the insertion succeeds (i.e., the key was not in the list) and `false` otherwise.

The insert initiates a call to find, in order to know where to link the newly created node (line 4). If the key does not exist, the operation allocates a new node out of a durable area using `allocFromArea()`. The allocation procedure (Section 5) returns a node that is available for use and whose validity state is valid, i.e., both validity bits have the same value. The insert operation then makes the node invalid by changing the first validity bit (line 12 Listing 4). The subsequent memory fence keeps the order between the writes and guarantees that the node becomes invalid before its initialization. This ensures that an incomplete node initialization will not confuse the recovery. Next, the operation initializes the node's fields, including the *next* pointer of the node (line 16), and then the operation tries to link the new node using a CAS (line 17). Note that the node is still invalid when linking it to the list. If the CAS fails, the entire operation is restarted and, if successful, the new node is made valid by flipping the second validity bit (line 18). It is then flushed to persist the insertion and `true` is returned.

If the key exists in the list, the existing node is made valid, then flushed and the operation returns `false` (lines 6 – 8). When finding a node with the same key, the existing node might not be valid yet because the node is linked to the list in an invalid state. It has to be made valid and persistent before `false` can be returned. Otherwise, a subsequent crash may reflect this failed insert but not reflect the preceding insert that caused this failure. This ensures durable linearizability.

The order between making the node valid and linking it is important. Making a node valid first and then linking it may cause inconsistencies. Consider a scenario with two threads trying to insert a node with a key k but with different values. Both threads may finish initializing their nodes and make them valid, but then the system crashes. During recovery, both nodes are found in a valid state (they may appear in the NVRAM even if an explicit flush was not executed), and there is no way to determine which should be in the set and which should not.

Listing 4. Link-Free List insert

```
1  bool insert(long key, long value){
2      while(true){
3          Node *pred, *curr;
4          pred, curr = find(key);
5          if(curr->key == key){
6              makeValid(curr);
7              FLUSH_INSERT(curr);
8              return false;
9          }
10
11         Node* newNode = allocFromArea();
12         flipV1(newNode);
13         atomic_thread_fence(memory_order_release);
14         newNode->key = key;
15         newNode->value = value;
16         newNode->next.store(curr, memory_order_relaxed);
17         if(pred->next.compare_exchange_strong(curr, newNode)){
18             makeValid(newNode);
19             FLUSH_INSERT(newNode);
```

```
20              return true;
21          }
22      }
23  }
```

3.4 The remove Operation

Given a key, the remove operation deletes the node with that key from the set. The return value is `true` when the removal was successful, i.e., there was such a node in the list, and now there is not, and `false` otherwise.

First, the requested node and its predecessor are found (line 5 Listing 5). If the node found does not contain the given key, the thread returns `false`. Otherwise, the node is made valid and then its *next* pointer is marked using a CAS (line 11). All along the code (and also here) we maintain the invariant that a marked node is valid. If the CAS succeeds, the operation finishes by calling `trim` to physically remove the node, and otherwise the removal is restarted.

There is no need for a `psync` operation between making `curr` valid (line 10) and the logical removal (line 11). Both modify the same cache line and the writes to the cache are ordered by the CAS (with default `memory_order_seq_cst`), implying the same order to the NVRAM. Therefore, the view of the node can be invalid and not marked (prior to line 10), valid and not marked (between lines 10 and 11), or valid and marked (after line 11). The node can never be in an inconsistent state (marked and invalid).

Listing 5. Link-Free List remove

```
1  bool remove(long key){
2      bool result = false;
3      while(!result){
4          Node *pred, *curr;
5          pred, curr = find(key);
6          if(curr->key != key)
7              return false;
8          Node* succ = getRef(curr->next.load());
9          Node* markedSucc = mark(succ);
10         makeValid(curr);
11         result = curr->next.compare_exchange_strong(succ, markedSucc);
12     }
13     trim(pred, curr);
14     return true;
15 }
```

3.5 Recovery

The validity scheme we use helps us determine whether a node was linked to the list before a crash occurred. This is possible because before initializing a node, it is made invalid so no partial writes are observed. If a remove operation manages to mark a node, we can know for sure it is removed.

The recovery takes place after a crash and the data it sees is data that was flushed to the NVRAM prior to the crash. The procedure starts by initializing an empty list with a head and a tail. Afterwards, it scans the durable areas of the threads for nodes. All nodes that are valid and unmarked are inserted, one by one, to an initially empty link-free list. All other nodes (invalid nodes and valid and marked nodes) are sent to the memory manager for reclamation. The linking

of the valid nodes is done without any psync operations since all the data in the nodes is already stored in the NVRAM.

4 THE DETAILS OF SOFT

The second algorithm we present is SOFT, whose number of psync operations matches the theoretical lower bound (of [Cohen et al. 2018]). It does so by dividing each update operation into two stages: intention and completion. In the intention stage, a thread triggers helping mechanisms by other threads, while not changing the logical state of the data structure. After the intention is declared, the operation becomes durable, in the sense that a subsequent crash will trigger a recovery that will attempt to execute the operation. In this section, we start by describing the nodes of the SOFT list (Sections 4.1 and 4.2), then we discuss the implementation details of each set operation (Sections 4.3, 4.4 and 4.5), and finally in Section 4.6, we explain the recovery scheme.

4.1 PNode

At the core of SOFT there is a *persistent node* (PNode) that captures the state of a given key in the NVRAM. It has a key, a value and three flags, which are described next. The structure of the persistent node is provided in Listing 6.

Listing 6. PNode

```
1 class PNode{
2     atomic<bool> validStart, validEnd, deleted;
3     atomic<long> key;
4     atomic<long> value;
5 } aligned(cache line size);
```

The PNode's three flags indicate the state of the node in the NVRAM. The first two flags have a similar meaning to the ones used by the link-free algorithm. When both flags are equal, the node is in a consistent state, and if the flags are different, then the node is in the middle of being inserted. It also has an additional flag indicating whether the node was removed.

Specifically, the PNode starts off with all three flags having the same value, *pInitialValidity*. In this case, the PNode is considered *valid* and *removed*. The negation of pInitialValidity is returned to the user of the node after calling alloc, and is denoted *pValidity*. From this point on, the state of the persistent node progresses by flipping the flags from pInitialValidity to pValidity.

When a key-value pair is inserted into the data structure, the corresponding PNode is made valid, by setting *validStart* to pValidity, assigning the key and the value of the node, and finally setting *validEnd* to pValidity. Only then, the persistent node is written to the NVRAM. When validStart differs from validEnd, the node is considered *invalid*. When validStart equals to validEnd (but is still different from deleted), the node is properly inserted and will be considered during recovery.

When the PNode is removed from the data structure, the *deleted* flag is set and the node is flushed. Then, the node is *valid* and *removed*, so it is not considered during recovery. Note that this represents exactly the same state as when the node was allocated, making the persistent node ready for future allocations. The only difference is the value of all flags, which was swapped from pInitialValidity to pValidity. Code for allocating, creating and destroying a PNode appears in Listing 7.

Listing 7. PNode Member Functions

```
1 bool PNode::alloc(){
2     return !validStart.load();
3 }
4
```

```
5  void PNode::create(long key, long value, bool pValidity){
6      validStart.store(pValidity, memory_order_relaxed);
7      atomic_thread_fence(memory_order_release);
8      this->key.store(key, memory_order_relaxed);
9      this->value.store(value, memory_order_relaxed);
10     validEnd.store(pValidity, memory_order_release);
11     psync(this);
12 }
13
14 void PNode::destroy(bool pValidity){
15     deleted.store(pValidity, memory_order_release);
16     psync(this);
17 }
```

4.2 Volatile Node

Volatile nodes have a key, a value, and a *next* pointer (to the next volatile node). In addition, they contain a pointer to a persistent node (i.e., a PNode, explained in Section 4.1) and *pValidity*, a Boolean flag indicating the pValidity of the persistent node. The structure of the volatile node appears in Listing 8.

Listing 8. Volatile Node

```
1  class Node{
2     long key;
3     long value;
4     PNode* pptr;
5     bool pValidity;
6     atomic<Node*> next;
7  };
```

Similar to the lock-free linked list algorithm by Harris [2001], the last bits of the *next* pointers store whether the node is deleted. Unlike Harris' algorithm, a volatile node must be in one of four states: "intention to insert", "inserted", "inserted with intention to delete", and "deleted", as discussed in the overview (Section 2.3). We assume standard methods for handling pointers with embedded state (lines 2 – 7 Listing 10). In addition, we use `trim` and `find` to physically unlink removed nodes and find the relevant window, respectively (Listing 9). Unlike its link-free counterpart, find also returns the state of both nodes. One is in the second address returned and the other is returned explicitly. Moreover, trim does not execute a psync before unlinking a node.

Listing 9. find and trim

```
1  bool trim(Node *pred, Node *curr) {
2     state predState = getState(curr);
3     Node *currRef = getRef(curr), *succ = getRef(currRef->next.load());
4     succ = createRef(succ, predState);
5     return pred->next.compare_exchange_strong(curr, succ);
6  }
7
8  Node*, Node* find(long key, state *currStatePtr){
9     Node *pred = head, *curr = pred->next.load();
10    Node *currRef = getRef(curr);
11    state predState = getState(curr), cState;
```

```
12    while (true){
13       Node *succ = currRef->next.load();
14       Node *succRef = getRef(succ);
15       cState = getState(succ);
16       if (cState != DELETED){
17          if (currRef->key >= key)
18             break;
19          pred = currRef;
20          predState = cState;
21       }
22       else
23          trim(pred, curr);
24       curr = createRef(succRef, predState);
25       currRef = succRef;
26    }
27    *currStatePtr = cState;
28    return pred, curr;
29 }
```

4.3 The contains Operation

The contains operation checks whether a key resides in the set. Unlike the insert and remove operations, contains is wait-free and does not use any psync operations.

A node is in the set only if its state is either "inserted" or "inserted with intention to delete". A node with the state "inserted with intention to delete" is still in the set because there is a thread trying to remove it, but it has not finished yet. Only in these two cases the return value is `true`; in all the other cases, it is `false`.

Listing 10. SOFT List contains

```
1  //Pseudo-code for managing state pointers
2  #def createRef(address, state) {.ptr=address, .state=state}
3  #def getRef(sPointer) {sPointer.ptr}
4  #def getState(sPointer) {sPointer.state}
5  #def stateCAS(sPointer, oldState, newState) {old=sPointer.load();
6      return sPointer.compare_exchange_strong(createRef(old.ptr, oldState),
7      createRef(old.ptr, newState));}
8
9  bool contains(long key){
10    Node *curr = head->next.load();
11    while (curr->key < key)
12       curr = getRef(curr->next.load());
13    state currState = getState(curr->next.load());
14    if(curr->key != key)
15         return false;
16     if(currState == DELETED || currState == INTEND_TO_INSERT)
17         return false;
18     return true;
19
20 }
```

4.4 The insert Operation

Insertion in SOFT follows the standard set API, which is getting a key and a value and inserting them into the set. The operation returns whether the insertion was successful. Code is provided in Listing 11 and is discussed below.

Similar to link-free, persistent nodes are allocated from a durable area using `allocFromArea`. When allocating a new PNode, all its validity bits have the same value, so its state is deleted. Volatile nodes can be allocated from the main heap.

The first step of insert is a call to `find`, which returns the relevant window (line 6). As mentioned above, while traversing the list, if a logically removed node, is found along the way the thread tries to complete its physical removal. Unlike link-free, however, there is no need to execute a `psync` a removed node before unlinking it. The volatile node becomes removed only after the corresponding PNode becomes removed and is written to the NVRAM. Therefore, if the state of a volatile node is "deleted", it is always safe to unlink it from the list and it does not require further operations.

Discovering a node with the same key already in the list fails the insertion. Nonetheless, the thread needs to help complete the insertion operation before returning, if the found node's state is "intention to insert". In the complementary case, when there is no node with the same key, the thread allocates a new PNode and a new volatile node, and attempts to link the latter node to the list (line 23) using a CAS. The new volatile node is initialized with the state "intention to insert", because we want other threads to help with finishing the insertion. If the CAS failed, the entire operation starts over. Otherwise, the thread moves to the helping part (lines 30 – 33), where the node is fully inserted.

The helping part starts by initializing the PNode of the appropriate node (line 30). Afterwards, all the threads try to complete the insertion and make it visible by changing the state of the new node to "inserted" (line 33). Finally, the thread returns `true` or `false` depending on the path taken.

Listing 11. SOFT List insert

```
1  bool insert(long key, long value){
2      Node *pred, *curr, *currRef, *resultNode;
3      state predState, currState;
4
5      while(true){
6          pred, curr = find(key, &currState);
7          currRef = getRef(curr);
8          predState = getState(curr);
9          bool result = false;
10         if(currRef->key == key){
11             if(currState != INTEND_TO_INSERT)
12                 return false;
13             resultNode = currRef;
14             break;
15         }
16         else{
17             PNode* newPNode = allocFromArea();
18             Node* newNode = new Node(key, value, newPNode, newPNode->alloc());
19             newNode->next.store(createRef(currRef, INTEND_TO_INSERT),
20                 memory_order_relaxed);
21
22             if(!pred->next.compare_exchange_strong(curr,
23                 createRef(newNode, predState)))
```

```
24                 continue;
25             resultNode = newNode;
26             result = true;
27             break;
28         }
29     }
30     resultNode->pptr->create(resultNode->key, resultNode->value,
31         resultNode->pValidity);
32     while(getState(resultNode->next.load()) == INTEND_TO_INSERT)
33         stateCAS(&resultNode->next, INTEND_TO_INSERT, INSERTED);
34
35     return result;
36 }
```

4.5 The remove Operation

The remove operation unlinks a node from the set with the same key as the given key. It returns `true` when the removal succeeds and `false` otherwise.

Similar to the previous operation, remove starts by finding the required window. If the key is not found in the set, the operation returns `false`. Recall that a volatile node is removed from the set only after its PNode becomes deleted in the NVRAM, so returning `false` is safe. Also, if the found node has a state of "intention to insert", the remove operation returns `false`. This is because such a node is not guaranteed to have a valid PNode in the NVRAM.

In the case when a node with the correct key is found, the thread attempts to mark the node as "inserted with intention to delete". At this point, all threads attempting to remove the node compete; the successful thread will return `true` while other threads will return `false` (line 14). This does not, however, change the logical status of the node (the key is still considered as inserted) or modify the NVRAM. Once the node is made "inserted with intention to delete", the thread calls `destroy` on the relevant PNode, so that the deletion is written to the NVRAM. Finally, the state is changed to be "deleted" to indicate the completion and the result is returned. Note that calling `destroy` and marking the node as "deleted" happens even if the thread fails in the "inserted with intention to delete" competition, in which case it helps the winning thread. The final step, executed only by the thread that won the "inserted with intention to delete" competition, physically disconnects the node from the list by calling `trim`. This latter step does not change the logical representation of the set and is executed only by a single thread to reduce contention.

Listing 12. SOFT List remove

```
1  bool remove(long key){
2      bool result = false;
3      Node *pred, *curr;
4      state predState, currState;
5      pred, curr = find(key, &currState);
6      Node* currRef = getRef(curr);
7      predState = getState(curr);
8      if(currRef->key != key)
9          return false;
10     if(currState == INTEND_TO_INSERT)
11         return false;
12
13     while(!result && getState(currRef->next.load()) == INSERTED)
```

```
14          result = stateCAS(&currRef->next, INSERTED, INTEND_TO_DELETE);
15      currRef->pptr->destroy(currRef->pValidity);
16      while(getState(currRef->next.load()) == INTEND_TO_DELETE)
17          stateCAS(&currRef->next, INTEND_TO_DELETE, DELETED);
18
19      if(result)
20          trim(pred, curr);
21      return result;
22  }
```

4.6 Recovery

In SOFT only the PNodes are allocated from the durable areas. All the volatile nodes are lost due to the crash. This means that the intentions are not available to the recovery procedure, so it decides whether a key is a part of the list based on the validity bits kept in the PNode. A PNode is valid and a part of the set, if the first two flags (`validStart` and `validEnd`) have the same value, and the last flag (`deleted`) has a different value.

In order to reconstruct the SOFT list, a new and empty list is allocated. Then the recovery iterates over the durable areas to find valid and not deleted PNodes. If such a PNode *pn* is found, a new volatile node *n* is allocated and its fields are initialized using the *pn*'s data. The value of *n*'s `pValidity` is set to the be *pn*'s `validStart`, and `pptr` points to *pn*. Finally, *n* is linked to the list in a sorted manner and its state is set to "inserted". Similar to link-free, no `psync` operations are used to link *n* since the data in *pn* already persisted in the NVRAM. Invalid or deleted PNodes are sent to the memory manager for reclamation.

5 MEMORY MANAGEMENT

Both of our algorithms use durable areas in which we keep the nodes with persistent data, which are used by the recovery procedure. A memory manager allocates new nodes and new areas, keeps record of old ones, and has free-lists for each thread. Moreover, since this is a lock-free environment, our algorithms are susceptible to the ABA problem [Michael 2002] and to use-after-free.

To maintain the lock-freedom of our algorithms, lock-free memory reclamation schemes can be used (e.g., [Alistarh et al. 2017; Balmau et al. 2016; Brown 2015; Cohen 2018; Cohen and Petrank 2015; Dice et al. 2016; Michael 2004]). Some, however, are complicated to incorporate; some require the data structure to be in a normalized form; and others have significant overhead that commonly deteriorates performance. We, therefore, chose to employ the very simple *Epoch Based Reclamation* scheme (EBR) [Fraser 2004] that is not lock-free but it performs very well and provides progress for the memory management when the threads are not stuck.

In EBR we have a global counter to indicate the current epoch, and each thread is either in an epoch (when executing a data structure operation) or *idle*. A thread joins the current epoch at the beginning of each operation, and becomes idle at its end. When an object is freed, it is added to a free-list for the current epoch. Whenever a thread runs out of memory, it starts the reclamation of the current epoch, denoted *e*. When all the threads reach either epoch *e* or an idle state, all the objects in the free-list related to epoch $e - 2$ can safely be reclaimed and reused. We used a variant of EBR that uses clock vectors. In particular, we used `ssmem`, an EBR that accompanies the ASCYLIB algorithms [David et al. 2015].

The `ssmem` allocator normally serves volatile memory, allocating objects of fixed predetermined size. We adapted it to our setting. In `ssmem`, each thread has its own personal allocator so the communication between different threads is minimal. The allocator provides an interface that allows allocating and freeing of objects of a fixed size in specially allocated designated areas. It

initially allocates a big chuck of memory from which it returns objects to the program using a bump pointer. When the area fills up, nodes get reclaimed, and holes emerge; a *free-list* is then used to allocate objects. Each thread has it own free-list so freeing nodes or using free ones does not require any form of synchronization. The free-lists are volatile and are reconstructed during a recovery. Invalid or deleted nodes a thread encounters during recovery while traversing the durable areas are inserted into the private free-list of the thread.

The memory manager keeps a list of all the areas it allocated so it can free them at the end of the execution. Throughout its life, the original `ssmem` manager does not free areas back to the operating system. In our implementation, empty areas can be returned to the operating system during the recovery if all the nodes of an area are free.

Both link-free and SOFT use durable areas as a part of their memory allocation scheme. These are address spaces in the heap memory that are used solely for node allocation and, therefore, `ssmem` can be used with small modifications. When a thread performs an insertion, it allocates a node from these areas, and when a node is removed, it is returned to the proper free-list. To reduce false sharing and contention, each thread has its own areas.

Using `ssmem`, each thread keeps a private list with one node per allocated area pointing to all the areas it allocated throughout the execution, denoted *area list*. This list has to be persistent so after a crash the areas will not be lost. We call nodes is this list *area nodes*. When allocating an additional area, we write its address in a new area node and write the new area node to the NVRAM. Then, we link it to the beginning of the area list (there is no need for any synchronization since the area list is thread-local), and flush the link to it, making the new area node persistent. The area list is persistent and its head is kept in a persistent thread-local space, which a recovery procedure can access. Thus, all the addresses of the different areas can be traced after a crash and all persistent nodes can be traversed.

There is an inherent problem when using durable algorithms without proper memory management. When inserting a new node, the node is allocated and only afterwards linked to the set. In the case of deletion, the node is unlinked from the set, and subsequently can be freed. Since a crash may occur at any time, we might have a persistent memory leak if a new node was not linked or if a deleted node was not freed.

Typically, this problem is solved by using a logging mechanism that records the intention (inserting or removing) along with the relevant addresses. This way, in case of a crash, the memory leaks may be fixed by reading the records. This logging mechanism requires more writes to the NVRAM, which take time, resulting in increased operation latency and worse throughput.

The durable areas solve this problem in a simpler manner since all the memory is allocated only from them. Therefore, when recovering and traversing the different areas, leaks will be identified using the validity scheme. Removed or invalid nodes can be freed and reused.

6 EVALUATION

We ran the measurements of the link-free and SOFT algorithms and compared them to the state-of-the-art set algorithm proposed by David et al. [2018]. We ran the experiments on a machine with 64 cores, with 4 AMD Opteron(TM) 6376 2.3GHz processors (16 cores each). The machine has 128GB RAM, 16KB L1 per one core, 2MB L2 for every pair of cores and 6MB LLC per 8 cores (half a processor). The machine's operating system is Ubuntu 16.04.6 LTS (kernel version 5.0.0). All the code was written in C++ 11 and compiled using g++ version 8.3.0 with a -O3 optimization flag.

NVRAM is yet to be commercially available, so following previous work [Arulraj et al. 2015; Ben-David et al. 2019; Chakrabarti et al. 2014; Cohen et al. 2019, 2017; David et al. 2018; Friedman et al. 2018; Kolli et al. 2016; Schwalb et al. 2015; Volos et al. 2011; Wang and Johnson 2014], we measured the performance using a DRAM. NVRAM is expected to be somewhat slower than DRAM

[Arulraj et al. 2015; Volos et al. 2011; Wang and Johnson 2014]. Nevertheless, we assume that data becomes durable once it reaches the memory controller[2]. Therefore, we do not introduce additional latencies to NVRAM accesses.

Link-free and SOFT use the `clflush` instruction to ensure that data is written back to the NVRAM (or to the memory controller). This instruction is ordered with respect to store operations [Intel 2019], so an additional store fence is not required (unlike the `clflushopt` instruction, which does require a fence). David et al. [2018] used a simulation of `clwb` (an instruction that forces a write back without invalidating the cache line, which is not supported by all systems). To compare apples to apples, we changed the code to execute a `clflush` instead (as other measured algorithms).

6.1 Throughput Measurements

We compared the algorithms to each other on three different fronts. Each test consisted of ten iterations, five seconds each and the results shown in the graphs, are the average of these iterations. In each test, the set was filled with half of the key range, aiming at a 50-50 chance of success for the insert and remove operations. Error bars represent 99% confidence intervals.

First, we measured the scalability of each algorithm, i.e., the outcome of adding more threads to increase the parallelism. The workload was fixed to 90% read operations (a common practice when evaluating sets [Herlihy and Shavit 2008]), and the key range was fixed as well. When running the lists, the key ranges were 256 and 1024. We chose to run two tests with the lists so we could have a closer look at the effect of a longer list on the scalability and performance. We also evaluated the hash set. For the hash set, we used a larger key range of 1M keys with a load factor of 1.

The results for the scalability test are displayed in Figure 1. On the left, the graphs show the throughput as a function of the number of threads (in millions of operations per second). On the right, the improvement relative to log-free set is depicted (the y axis is the improvement factor).

In Figures 1a and 1b, we can see the results for the shorter and longer lists. When the key range is 256 keys, all algorithms experience a peak with 16 threads and a slow decrease towards 64 threads. For a single thread, SOFT and link-free outperform log-free by 40% and 35%, respectively, for 16 threads by 30% and 20%, respectively, and for 64 threads, both by 94%. The 16-thread peak can be explained by the nature of a list. Running many threads on a short list implies contention that hurts performance. Also, 16 threads can use a single processor but 17 cannot.

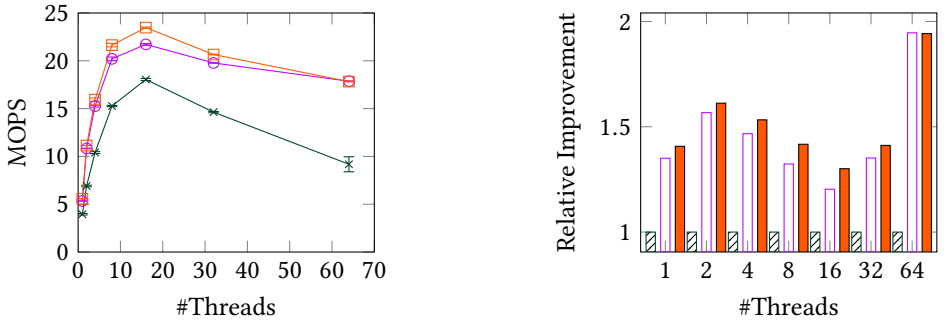
SOFT achieves the best performance on the short list by a noticeable margin. In this case, the amount of `psync` operations dominates performance as the traversal times are short. Unlike link-free or log-free, SOFT uses the optimal number of fences per update. For instance, both link-free and log-free executed a `psync` before trimming a logically deleted node (SOFT does not). Both of our algorithms perform much better than log-free and we can relate this result to the elimination of pointer flushing, which is the main idea behind both algorithms.

For a longer list (Figure 1b), all the compared lists scale with the additional threads. When the number of available keys is bigger, most of the time is spent on traversing the list; hence, more threads imply more concurrent traversals and more operations.

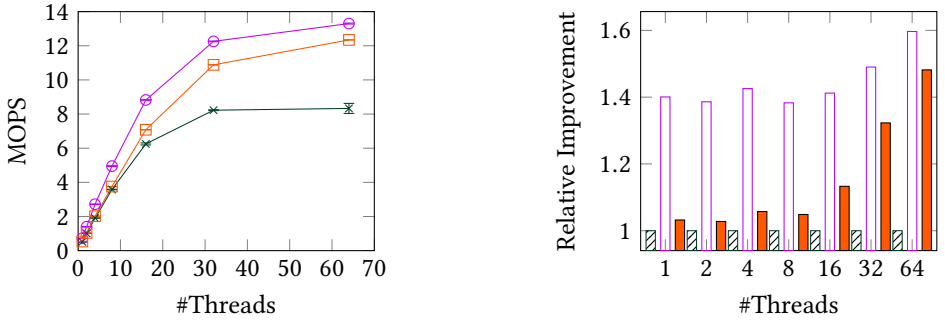
As can be seen in the graph, link-free outperforms both SOFT and log-free by a considerable difference. In contrast to Figure 1a, here the additional overhead of SOFT (using intermediate states and more CAS-es instead of direct marking) degrades its performance. When the range grows, the additional `psync` operations are masked by the traversal times. Since SOFT uses two additional CAS-es in each update, link-free wins.

Moreover, with higher contention, a node might be flushed more than once in link-free. As mentioned, link-free prevents redundant `psync` operations using a flag after the first necessary

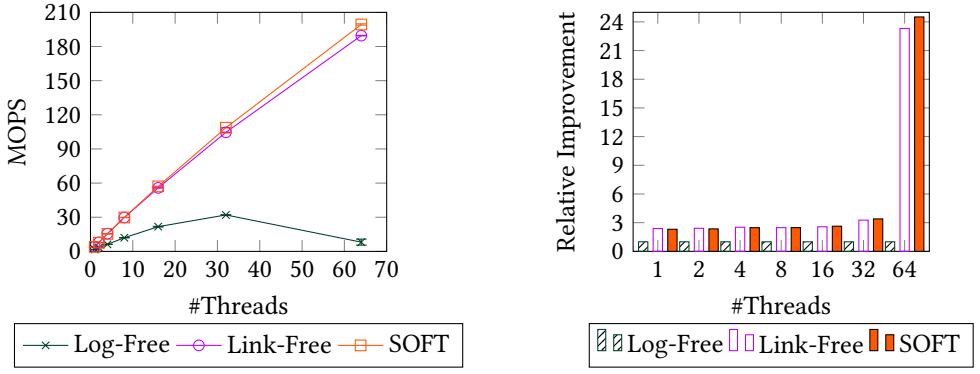
[2] https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction



(a) List's Throughput with Range of 256



(b) List's Throughput with Range of 1024

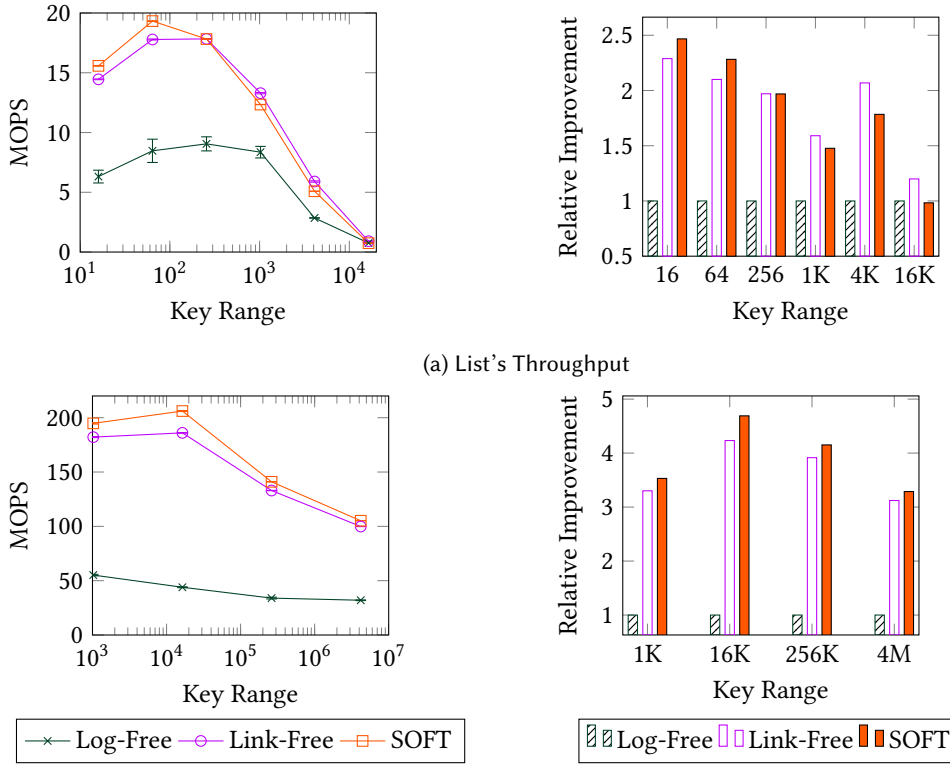


(c) Hash Table's Throughput with Range of 1M

Fig. 1. Throughput as a Function of the #Threads

psync. In a case where multiple threads operate on the same key, it might be flushed more than needed. So, when contention is high, link-free may perform more psync operations. For cases of lower contention, the optimization is more effective. In effect, link-free does a single psync per update and zero per read (due to the low contention, all flags are set before other threads help). In this case, link-free and SOFT execute the same amount of psync operations, but SOFT is more complicated and uses more CAS-es. Because of this, for boarder ranges, link-free performs better.

The hash set is evaluated in Figure 1c. Link-free and SOFT are highly scalable (reaching 25.2x and 27x with 32 threads, respectively, and 45.6x and 49.6x with 64 threads, respectively). Log-free is a



(a) List's Throughput

(b) Hash Tables' Throughput

Fig. 2. Throughput as a Function of Key Range

lot less scalable (18.4x with 32 threads and 4.6x with 64 threads). For 32 threads, SOFT and link-free perform better by factors of 3.4x and 3.26x, respectively. Thus, we obtain a dramatic improvement of the state-of-the-art.

As can be seen, the result of the log-free hash table in the test with 64 threads is oddly low. We used the authors' implementation and we do not know why this happened. To make further comparisons fair enough to previous work, we fixed the number of threads at 32 in subsequent hash table evaluations. The number of threads in the lists' evaluation remained 64.

In the second experiment, we examined the effect of different key ranges on the performance of the data structure. We again fixed the workload to be 90% read operations, and the number of threads at 64 for the lists and at 32 for the hash maps. The sizes when running the lists vary from 16 to 16K in multiples of 4. For hash tables, the size varies between 1K and 4M in multiples of 16.

Figure 2a shows that SOFT and link-free are superior to log-free in each key range. As expected, for shorter ranges, SOFT performs better and for bigger ranges link-free wins. The reason is that as the key range grows, more time is spent on traversals of the lists and the number of psync operations used is masked. We can see this effect in the graph: as the range grows, the difference in performance shrinks, starting with a factor of 2.46x difference between SOFT and log-free and ending with link-free having a 20% improvement for 16K keys.

As expected, the trend of the graph consists of a single peak point. We note that the performance improves because contention drops when the range grows but only up to a point. Beyond this point, most of the time is spent on traversing the list rather than executing actual operations.

Figure 2b depicts the performance of the three hash tables and the improvement relative to log-free. As explained above, this test was run with 32 threads. As predicted, the performance of all hash tables worsens as the range grows. This may be attributed to reduced locality. For 1K distinct keys, SOFT outperforms log-free by a factor of 3.53x and link-free outperforms log-free by a factor of 3.2x. For the longest range (4M keys), SOFT is better by a factor of 3.28x and link-free is better by a factor of 3.12x.

The last variable evaluated is the workload. We measured different distributions of reads (50% – 100% with increments of 10%, and also the specific values of 95%). Note that this covers the standard "Yahoo! Cloud Serving Benchmark" (YCSB) [Cooper et al. 2010] workloads A (50% reads), B (95% reads), and C (100% reads). In this experiment, the number of threads was fixed at 64 for lists and 32 threads for hash tables, and the key ranges were fixed at 256 or 1024 in the case of the lists and at 1M in the case of the hash tables.

The lists (Figures 3a and 3b) all behaved similarly to one another. For both ranges, link-free performed slightly better than SOFT. Link-free is superior to SOFT since the high amount of threads increases the contention, which increases the cost of the additional CAS-es used in SOFT. Also, a higher percentage of updates also contributed to more CAS-es in SOFT.

For the shorter range, link-free surpassed log-free by a factor of 2.6x with 50% reads, and for 100% reads, it had a 33% improvement. With 1k keys, the throughput of link-free was higher by a factor of 2.1x with 50% reads and higher by 23% with 100% reads.

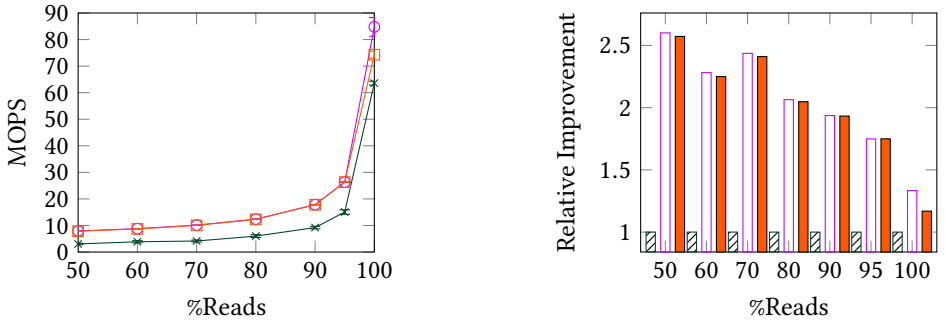
The trend of both graphs can be justified by a few reasons. First, all algorithms use the least amount of psync operations in the read operations. SOFT does not use any, link-free uses at most one, and log-free uses at most two. Moreover, reads are faster since there is no need to invalidate any cache lines of other processors. Finally, unlike insert and remove, which may restart and theoretically run forever, the contains operation is wait-free and optimized to run as fast as possible. Accordingly, the gap between the different algorithms shrinks as the percentage of reads grows.

Running with 100% reads is a special situation where the performance improves tremendously. Each thread runs in isolation from the others since there are no conflicts between contains operations. Also, in this case, none of the algorithms execute any psync operations. Link-free and log-free both use optimizations to reduce the number of psync operations and since the nodes in the list were inserted and flushed previous to the beginning of the test, there is no need to flush them again.

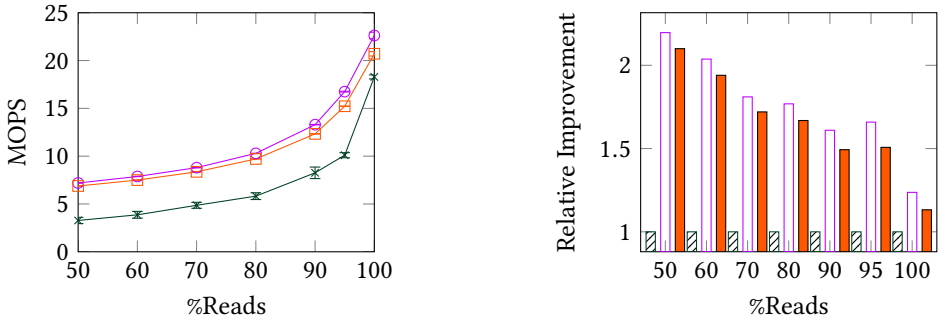
We would expect SOFT to be the best in this scenario but due to its implementation, it falls short. Unlike link-free, each volatile node in SOFT has an additional pointer that makes it larger. As a result, about one and a half volatile nodes fit in a single cache line, so when traversing the list, we have more cache misses. SOFT is still better than log-free because its contains operation is simpler. Log-free has a few branches to check whether a node should be flushed or not, which lengthens the function and may cause branch mis-predictions.

The hash tables, depicted in Figure 3c, exhibit a trend similar to what we saw in previous tests. The throughput rises as the number of updates declines. Moreover, the difference in performance between the three algorithms shrinks as the number of updates decreases.

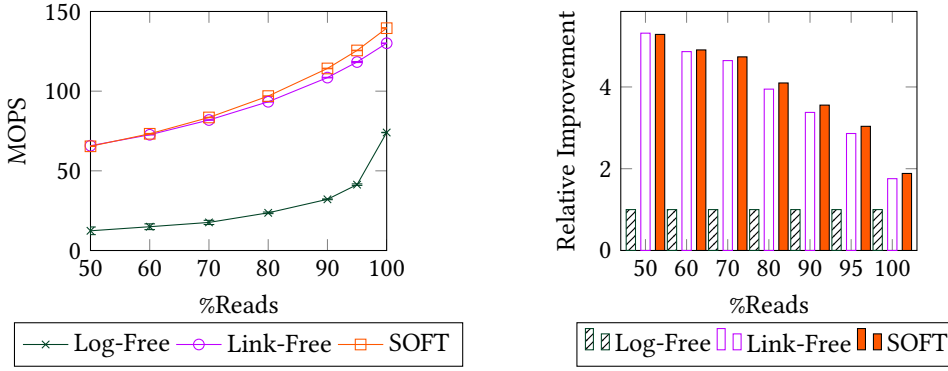
In according with our expectations, SOFT surpasses link-free and log-free. The traversal times in the hash tables are minimal so SOFT does not suffer from cache misses and the simplistic contains operation works in SOFT's favor.



(a) List's Throughput with Range of 256



(b) List's Throughput with Range of 1024



(c) Hash Table's Throughput with Range of 1M

Fig. 3. Throughput as a Function of the Percentage of Reads

7 RELATED WORK

There has been a lot of research focused on adapting specific concurrent data structures to durable ones [David et al. 2018; Friedman et al. 2018; Nawab et al. 2017; Schwalb et al. 2015]. Some researchers developed techniques to modify general objects into durable linearizable ones [Avni and Brown 2016; Coburn et al. 2012; Cohen et al. 2018; Izraelevitz et al. 2016; Kolli et al. 2016; Volos et al. 2011]

Coburn et al. [2012]; Kolli et al. [2016]; Volos et al. [2011] used transactions to create a new interface to the NVRAM and, by proxy, make regular objects durable linearizable. The main disadvantage of their schemes is the need to log operations and other kinds of metadata in the NVRAM, which causes more explicit writes to the memory and uses of synchronization primitives. Another major disadvantage is the use of locks that limits the scalability of the different implementations and might cause an unbounded rollback effect upon a crash.

Izraelevitz et al. [2016] presented a general algorithm to maintain durable linearizability. This generality, however, comes at the expense of efficiency; their construction inserts a fence before every shared write and a flush after, a fence and a psync for each CAS, and a psync after every shared read. In contrast, our algorithms are optimized in the sense they execute fewer psync operations, especially SOFT.

Cohen et al. [2017] presented a sequential durable hash table that uses only one psync per update and none for reads, achieving the lower bound proven by Cohen et al. [2018]. This paper introduced the validity schemes we used in both algorithms. Both algorithms rely on the observation made in the paper that the order of writes to the *same cache line* in the program is the same as the order of those writes in the memory. No extension to concurrency was discussed in their paper.

Nawab et al. [2017] developed an efficient hash table that supports multiple threads and transactions. They used fine-grained synchronization, and thus their algorithm is not lock-free. Their algorithm does not support durable linearizability but only buffered durable linearizability which is a weaker guarantee. Thus this work is not comparable to ours.

Friedman et al. [2018] presented three variations of a durable lock-free queue. The first guarantees durable linearizability [Izraelevitz et al. 2016], the second guarantees *detectable execution* [Friedman et al. 2018], which is a stronger guarantee than durable linearizability, and the third guarantees buffered durable linearizability [Izraelevitz et al. 2016]. The queue is inherently different from a set since it maintains an order between individual keys.

Cohen et al. [2018] introduced a theoretical universal construct to obtain durable lock-free objects with one psync per update (per conflicting thread) and none for reads. Their implementation uses a lock-free queue to order all pending operations, then a batch of operations is persisted together and, finally, a flag is set to indicate that the operations were flushed. This algorithm is theoretical and is not targeted at high performance. Using a queue to order operations creates contention and hurts scalability. In addition, the state of the object is a persistent log of all the previous operations, which means that in order to return a result, the whole log has to be traversed, making this algorithm highly inefficient and impractical.

David et al. [2018] introduced four kinds of sets (*Log-Free Data Structures*), building up from lock-free data structures and adding to them two main optimizations. *Link-and-persist* is the first optimization and it reduces the number of psync operations but at the cost of using CAS, which is considered more expensive than a simple `store` operation [David et al. 2013]. The second is *link-cache*, which writes *next* pointers to the NVRAM only when another operation depends on the persistency of the pointer. This work represents state-of-the-art durable sets and we compared our constructions to it, showing dramatic improvements.

8 CONCLUSION

In this work we presented two algorithms for durable lock-free sets: link-free and SOFT. These two algorithms were shown to outperform existing state-of-the-art by significant factors of up to 3.3x. In addition to high efficiency, they also demonstrated excellent scalability. The main idea underlying these algorithms was to avoid persisting the data structure's pointers, at the expense of reconstructing the data structures during (infrequent) recoveries from crashes. SOFT reduces fences to the minimum theoretical value, at the expense of algorithmic complication and higher (volatile)

synchronization. The evaluation demonstrated that SOFT outperforms the link-free implementation when psync operations are often required: For example, for long lists it was better to use the link-free version because traversals were long and psync operations were infrequent. For short lists (which also underlay a hash table), however, operations are short and psync operations occur frequently. In this case, SOFT was the best performing method.

REFERENCES

Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 483–498. https://doi.org/10.1145/3064176.3064214

Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 707–722. https://doi.org/10.1145/2723372.2749441

Hillel Avni and Trevor Brown. 2016. Persistent Hybrid Transactional Memory for Databases. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 409–420. https://doi.org/10.14778/3025111.3025122

Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, New York, NY, USA, 349–359. https://doi.org/10.1145/2935764.2935790

Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA '19)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/3323165.3323187

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*. ACM, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. https://doi.org/10.1145/2714064.2660224

Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2012. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 47, 4 (2012), 105–118.

Nachshon Cohen. 2018. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 143.

Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

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

Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/3210377.3210400

Nachshon Cohen and Erez Petrank. 2015. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 254–263. https://doi.org/10.1145/2755573.2755579

Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 514–530. https://doi.org/10.1145/2983990.2983995

Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 373–386. https://www.usenix.org/conference/atc18/presentation/david

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 33–48. https://doi.org/10.1145/2517349.2522714

Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1414–1425. https://doi.org/10.14778/1920841.1921015

Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-intrusive Memory Reclamation for Highly-concurrent Data Structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 36–45. https://doi.org/10.1145/2926697.2926699

Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-lists. In *Distributed Computing*, Jennifer Welch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–314.

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2006. A Lazy Concurrent List-based Set Algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05)*. Springer-Verlag, Berlin, Heidelberg, 3–16. https://doi.org/10.1007/11795490_3

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

Intel 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Intel.

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *J. Emerg. Technol. Comput. Syst.* 12, 1, Article 8 (Aug. 2015), 19 pages. https://doi.org/10.1145/2700249

Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. *ACM SIGPLAN Notices* 51, 4 (2016), 399–411.

Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 8:1–8:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.8

Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133920

Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 129–144. https://www.usenix.org/conference/osdi18/presentation/maeng

Maged M. Michael. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256

Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Andréa W. Richa (Ed.), Vol. 91. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 37:1–37:16. https://doi.org/10.4230/LIPIcs.DISC.2017.37

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 497–514. https://doi.org/10.1145/3132747.3132765

Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy-harvesting Computing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1085–1100. https://doi.org/10.1145/3314221.3314583

David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics (IMDM '15)*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2803140.2803144

Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)* 53, 3 (2006), 379–405.

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

Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation without Hardware Support or Programmer Intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 17–32. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude

Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18)*. ACM, New York, NY, USA, 41–53. https://doi.org/10.1145/3274783.3274837

Y. Zhang and S. Swanson. 2015. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–10. https://doi.org/10.1109/MSST.2015.7208275