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), 47 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 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.

[*]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.

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

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 our GitHub repository).

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.

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`

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

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

```
class Node{
    atomic<byte> validityBits;
    atomic<bool> insertFlushFlag, deleteFlushFlag;
    long key;
    long value;
    atomic<Node*> next;
} 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

```
bool trim(Node *pred, Node *curr){
   FLUSH_DELETE(curr);
   Node *succ = getRef(curr->next.load());
   return pred->next.compare_exchange_strong(curr, succ);
}

Node*, Node* find(long key){//method returns two pointers, pred and curr.
    Node* pred = head, *curr = head->next.load();
    while(true){
        if(!isMarked(curr->next.load())){
            if(curr->key >= key)
                break;
            pred = curr;
        }
        else
            trim(pred, curr);
        curr = getRef(curr->next.load());
    }
    return pred, curr;
}
```

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          }
```

```
    }
}
```

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

```
class PNode{
    atomic<bool> validStart, validEnd, deleted;
    atomic<long> key;
    atomic<long> value;
} 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

```
bool PNode::alloc(){
    return !validStart.load();
}

void PNode::create(long key, long value, bool pValidity){
    validStart.store(pValidity, memory_order_relaxed);
    atomic_thread_fence(memory_order_release);
    this->key.store(key, memory_order_relaxed);
    this->value.store(value, memory_order_relaxed);
    validEnd.store(pValidity, memory_order_release);
    psync(this);
}

void PNode::destroy(bool pValidity){
    deleted.store(pValidity, memory_order_release);
    psync(this);
}
```

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

```
class Node{
   long key;
   long value;
   PNode* pptr;
   bool pValidity;
   atomic<Node*> next;
};
```

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

```
bool trim(Node *pred, Node *curr) {
   state predState = getState(curr);
   Node *currRef = getRef(curr), *succ = getRef(currRef->next.load());
   succ = createRef(succ, predState);
   return pred->next.compare_exchange_strong(curr, succ);
}

Node*, Node* find(long key, state *currStatePtr){
   Node *pred = head, *curr = pred->next.load();
   Node *currRef = getRef(curr);
   state predState = getState(curr), cState;
   while (true){
      Node *succ = currRef->next.load();
      Node *succRef = getRef(succ);
      cState = getState(succ);
      if (cState != DELETED){
         if (currRef->key >= key)
            break;
         pred = currRef;
         predState = cState;
      }
      else
         trim(pred, curr);
      curr = createRef(succRef, predState);
      currRef = succRef;
   }
   *currStatePtr = cState;
   return pred, curr;
}
```

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

```
//Pseudo-code for managing state pointers
#def createRef(address, state) {.ptr=address, .state=state}
#def getRef(sPointer) {sPointer.ptr}
#def getState(sPointer) {sPointer.state}
#def stateCAS(sPointer, oldState, newState) {old=sPointer.load();
    return sPointer.compare_exchange_strong(createRef(old.ptr, oldState),
    createRef(old.ptr, newState));}

bool contains(long key){
   Node *curr = head->next.load();
   while (curr->key < key)
      curr = getRef(curr->next.load());
   state currState = getState(curr->next.load());
   if(curr->key != key)
       return false;
    if(currState == DELETED || currState == INTEND_TO_INSERT)
       return false;
    return true;

}
```

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)))
```

```
                continue;
            resultNode = newNode;
            result = true;
            break;
        }
    }
    resultNode->pptr->create(resultNode->key, resultNode->value,
        resultNode->pValidity);
    while(getState(resultNode->next.load()) == INTEND_TO_INSERT)
        stateCAS(&resultNode->next, INTEND_TO_INSERT, INSERTED);

    return result;
}
```

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)
```

```
        result = stateCAS(&currRef->next, INSERTED, INTEND_TO_DELETE);
    currRef->pptr->destroy(currRef->pValidity);
    while(getState(currRef->next.load()) == INTEND_TO_DELETE)
        stateCAS(&currRef->next, INTEND_TO_DELETE, DELETED);

    if(result)
        trim(pred, curr);
    return result;
}
```

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.

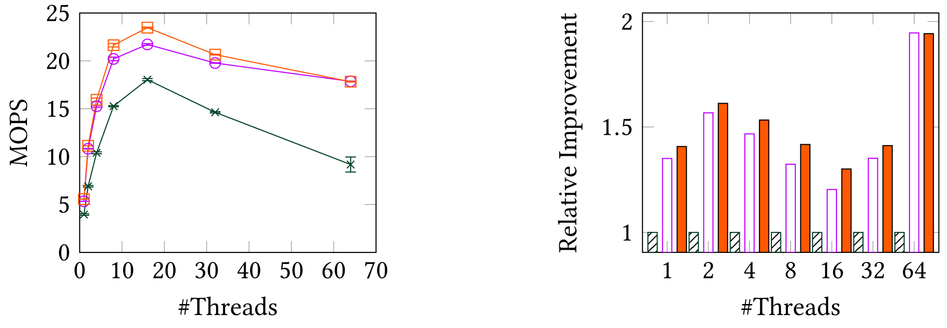
SOFT achieves the best performance on the short list by a noticeable margin. In this case, the amount of `psync` operations dominates performance as the traversal times are short. Unlike link-free or log-free, SOFT uses the optimal number of fences per update. For instance, both link-free and log-free 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.

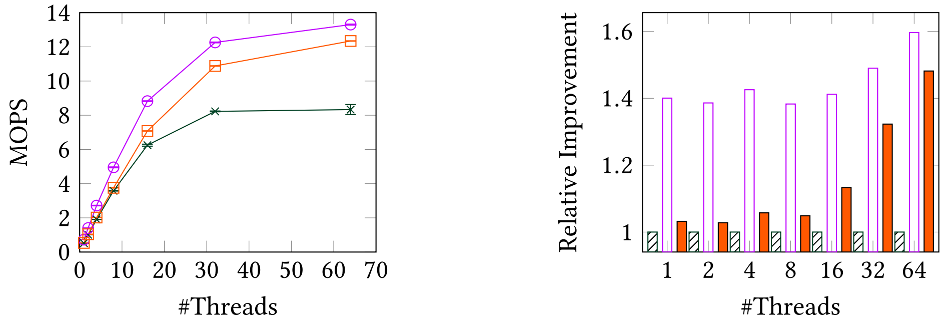
As can be seen in the graph, link-free outperforms both SOFT and log-free by a considerable difference. In contrast to Figure 1a, here the additional overhead of SOFT (using intermediate states and more CAS-es instead of direct marking) degrades its performance. When the range grows, the additional `psync`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

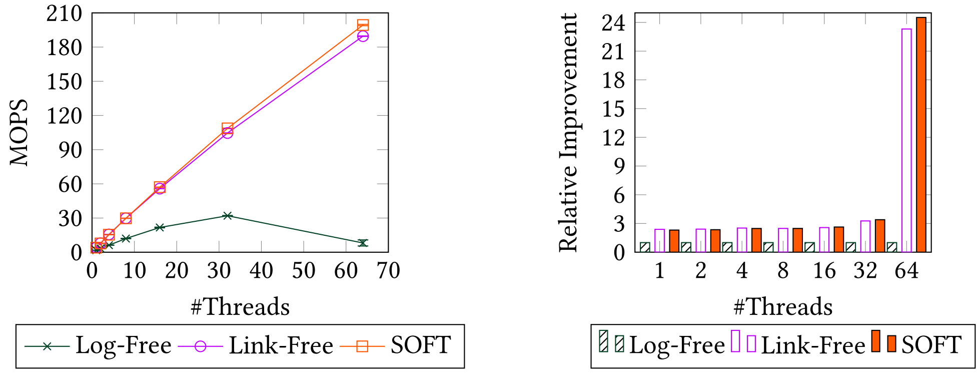
[2]https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction



(a) List's Throughput with Range of 256



(b) List's Throughput with Range of 1024

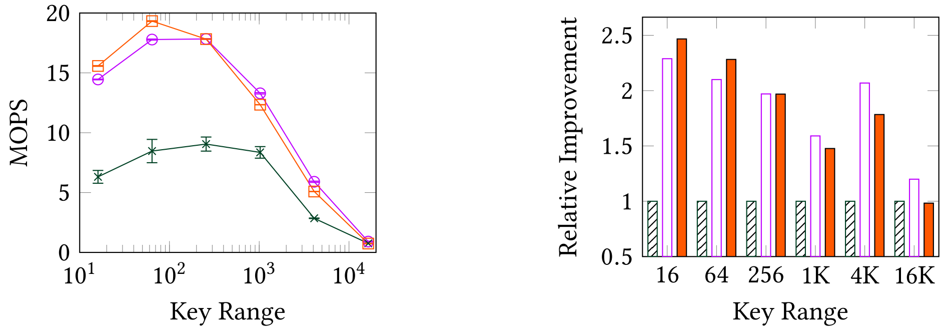


(c) Hash Table's Throughput with Range of 1M

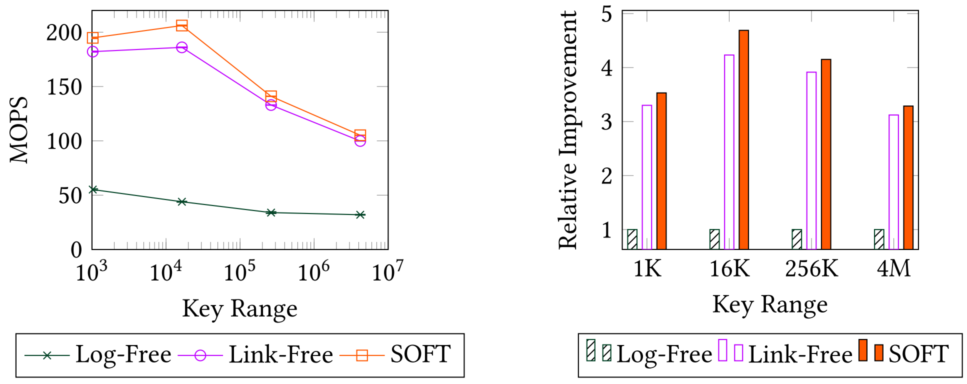
Fig. 1. Throughput as a Function of the #Threads

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.

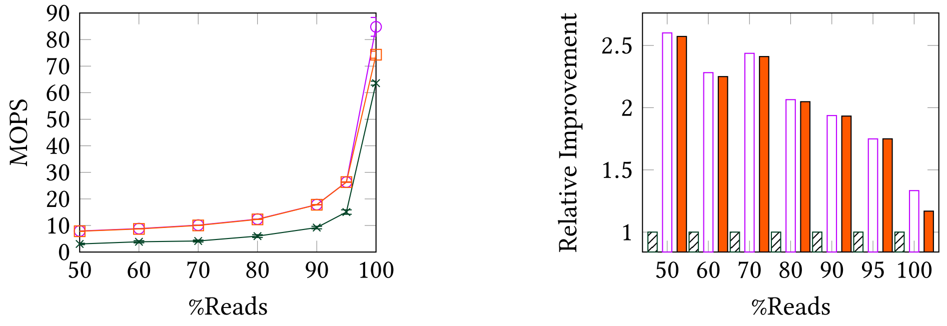
The trend of both graphs can be justified by a few reasons. First, all algorithms use the least amount of psync operations in the read operations. SOFT does not use any, link-free uses at most one, and log-free uses at most two. Moreover, reads are faster since there is no need to invalidate any cache 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.

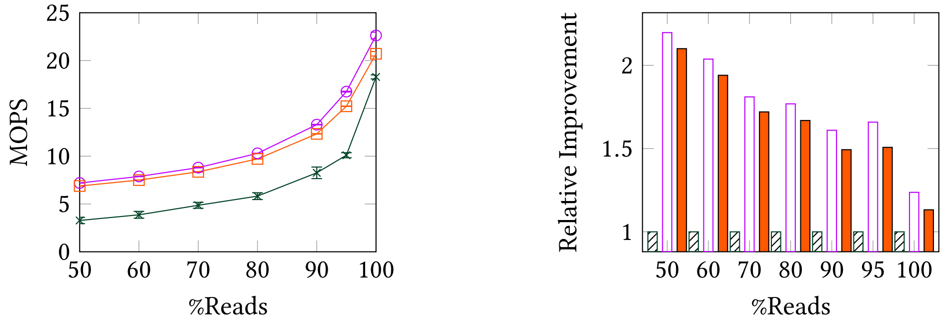
We would expect SOFT to be the best in this scenario but due to its implementation, it falls short. Unlike link-free, each volatile node in SOFT has an additional pointer that makes it larger. As a result, about one and a half volatile nodes fit in a single cache line, so when traversing the list, we 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.

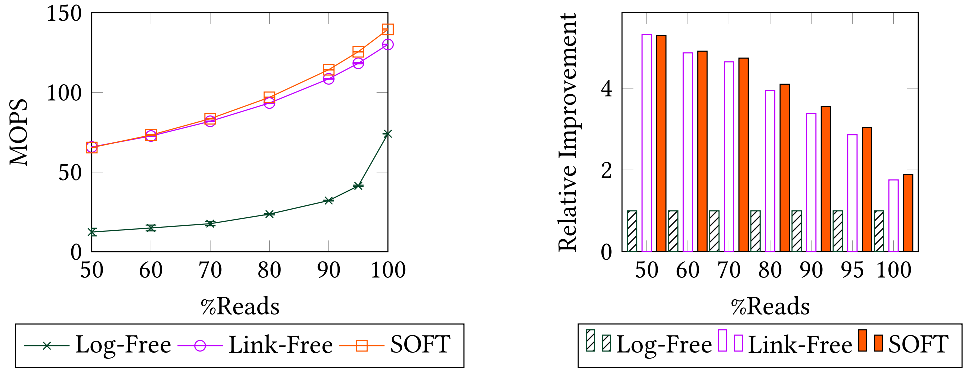
In according with our expectations, SOFT surpasses link-free and log-free. The traversal times in the hash tables are minimal so SOFT does not suffer from cache misses and the simplistic contains operation works in SOFT's favor.



(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 &#38; 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

Nancy A Lynch. 1996. *Distributed algorithms.* Elsevier.

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

A PROOF TERMINOLOGY

In this section we present the terminology we use in order to prove that the implementations of the link free and soft lists are correct. The proof for the link-free list appears in Section B and the proof for the soft list appears in Section C. We assume a sequentially consistent execution in both proofs, i.e., we do not consider reordering of loads and stores in our proof. This simplifies the proofs and is also unavoidable because there is no formal definition of C++11 extended to include psync operations.

We use the notion of *durable linearizability* [Izraelevitz et al. 2016] for correctness. To show that the proposed lists are durable linearizable, we define linearization points for all of the completed operations (either by returning or after a recovery procedure), as well as some of the operations which are still pending during a crash event. Note that in our setting, following a crash, recovery is always completed before any new operation begins executing, so there exists a point in time where recovery is completed and no new operation began executing.

We consider a standard shared memory setting with a bounded collection of threads, communicating via shared memory space [Lynch 1996]. Data structures are accessed via their operations. A data structures includes a (finite or infinite) set of possible configuration (with a distinguished initial configuration) and a sequential specification. The *sequential specification* defines the expected behavior of the data structure in a sequential execution. It specifies, per operation, the preconditions and their respective postconditions. It also takes into account the operation's input parameters. We consider deterministic data structures for which, given an operation and a current configuration (that obeys the operation's preconditions), there exists a single resulting configuration that satisfies the operation's postconditions.

A *configuration* at any point during the execution *state* is an instantaneous snapshot of the system, specifying the content of the shared memory space as well as local variables and state for each thread. In the *initial configuration* the shared data structure and the local variable are all in their initial state, and each thread's program counter is the set to the beginning of its program.

The threads change the system's configuration by taking *steps*, which may include performing some local computations and, additionally, an operation on the shared memory. In particular, an invocation of an operation (together with the respective input parameters) and the return from an operation are each considered to be a single step. We assume each step is atomic. An *execution* consists of an alternating sequence of configurations and steps, starting with the initial state. These steps are called *events*. We often consider only the execution steps, since the configurations can be easily computed given the execution steps. Each step is coupled with the executing thread (its identity depends on the *scheduler*). If it is an invocation of an operation, it also includes the input parameters, and if it is the last step of the operation, it is associated with the operation result. This last step is denoted the *return* or the *response* of the operation. A sub-execution of a given execution E is a sub-sequence of events in E.

A.1 Linearizability

For each invocation of an operation in an execution we can match a response that follows it in the execution with the same invoking thread and operation type. We can also match execution steps of this operations (by the same thread) in between the two. A invocation is called *pending* in execution E if its matching response does not exist. We denote *complete(E)* as a sub-execution of E containing all pairs of matching invocations and responses, and all execution steps executing these operations. For each thread T we define a sub-execution by using the notation $E|T$. This sub-execution is the sub-sequence of steps of operations performed by thread T. Two executions, $E, \hat{E}$ are equivalent if for every thread T they are the same, $E|T = \hat{E}|T$. A execution S is called *sequential* if it begins with an invocation and a new invocation in the execution occurs immediately after the response to the previous invocation occurred. We call a execution *well-formed* if for every thread T, $E|T$ is sequential.

We say that an operation op_0 *happens before* an operation op_1 in a execution E, if the response of op_0 occurs in E before op_1's invocation. In this case, we write $op_0 \prec_E op_1$. The relation *happens before* defines a partial order of a execution E.

Sequential specification. Given a data structure, we define a *sequential specification* as a set of sequential executions that are legal for this data structure.

Definition A.1 (Linearizability [Herlihy and Wing 1990]). A execution E of data structure operations is *linearizable* if it can be extended by adding zero or more responses at the end of E to create $\hat{E}$, and there exists a legal sequential execution S for the data structure such that:

(1) $complete(\hat{E})$ is equivalent to S
(2) if $m_0 \prec_{\hat{E}} m_1$, then $m_0 \prec_S m_1$ in S.

We refer to S as the linearization of E, and the order of the operations in S is called the linearization order.

Informally, given a execution of invocations, some of which might be pending, and responses, a execution is *linearizable* if there exists a *linearization order* L of some of the operations in E that satisfies the following. All complete operations appear in L, where complete means that the operation's invocation has a matching response in the execution. Some pending invocations are completed by the responses added in $\hat{E}$ and these must also appear in L. The remaining pending invocations do not appear in L. If we execute the operations according to their linearization order sequentially, we get the same execution (equivalent parameters and results), and the sequential execution is legal for the data structure.

Often, papers use an equivalent definition, in which the second condition is replaced as follows. We associate a *linearization point* for each operation with a step in the execution, and require that the linearization point of each operation appears between its invocation and its response. The linearization order is then determined by the order of the linearization points. It is easy to see that adequate linearization points satisfy the second condition. Showing that adequate linearization points can be specified for each adequate linearization is also not hard.

In this paper we sometimes view a linearization of an execution as a specification of linearization points and at other times as an order of the operations that satisfies Condition (2). These views are equivalent.

A.2 Durable Linearizability

Moving to durable linearizability, we add crashes as new types of events in a execution and a new correctness criterion is required. We use the notion of *durable linearizability* [Izraelevitz et al. 2016] for correctness. In the setting of durable linearizability, after a crash, new threads are created to continue the tasks of the program. A recovery procedure is executed after each crash by the new threads to complete some of the pending operations that were executed before the crash. The recovery is completed before new operations start executing. Loosely speaking, we require that the execution, minus all crash events and minus operations that did not survive the crashes, be linearizable. Given an execution E of data structure operations, we denote $ops(E)$ the sub-execution where all the crashes are omitted.

Definition A.2 (Durable Linearizability). A execution E is said to be *durably linearizable* if it is well-formed (i.e., for every thread T, $E|T$ is sequential) and ops(E) is linearizable. A data structure is durable linearizable if any concurrent execution of data structure operations yields a durably linearizable execution.

Note that after a crash, new threads are generated to continue the program execution. Therefore, any operation that executes concurrently with a crash is a pending operation that can never complete. It cannot have a matching response in the execution because the thread that invoked it does not execute beyond the crash. In contrast, any operation that completes before the crash has a matching response that completes it. Definition A.2 requires a linearization of all operations in the execution. By Definition A.1 (linearizability), this means that all completed operations must be included in the linearization order, but operations that executed concurrently with a crash are pending at the end of the execution and they do not need to be in the linearization order. Some of these operations may be matched with a response at the end of the execution and be included in the linearization order, whereas other operations may remain pending and be left out of the linearization order.

In our proof, we specify a linearization order for executions of the proposed durable lists. We specify which of the operations that execute concurrently with the crash survive, i.e., are matched with a response at the end of the execution and are included in the linearization order. We denote these operations as operations that *survive* the crash.

The recovery procedure, executed after a crash (and described in Section 3.5 and 4.6), is assumed to terminate before new threads start executing their code. Given an operation for which a crash event occurs after its invocation and before its response, we consider its response point as the end of the respective recovery procedure. Notice that in the following definitions, we do not consider recoveries that are interrupted by crash events. We do so for clarity and brevity. The definitions can be easily extended to include such cases.

B LINK-FREE CORRECTNESS

We start by proving some basic list invariants. In Section B.1 we prove the linearizability of our implementation when there are no crash events, and that it is also durable linearizable. Finally, we show that our implementation is lock-free in Section B.2.

The content of a node in the volatile memory, can be different from its content in the NVRAM, due to modifications that have not been persisted yet (either by implicit or explicit flushes). We distinguish between the two representations of a single link-free node: the *volatile node*, and the *persistent copy* which contains only the modifications written back to the NVRAM (implicitly or explicitly).

We start by stating some basic definitions we are going to use throughout our proof. Notice that, unless stated otherwise, the definitions relate to the volatile nodes (regardless of being written to the non-volatile memory).

Definition B.1 (Reachability). We say that a node n is *reachable* from a node n' if there exists nodes $n_0, n_1, \ldots, n_k$ such that $n_0 = n'$, $n_k = n$ and for every $0 \leq i < k$, n_i is the predecessor of n_{i+1} (via its *next* pointer). We say that a node n is *reachable* if it is reachable from the `head` sentinel node.

Definition B.2 (Infant Nodes). We say that a node n is an *infant* if n is neither `head` nor `tail`, and there does not exist an earlier successful execution of the CAS operation in line 17 of Listing 4, satisfying $newNode = n$.

Definition B.3 (A Node's State). Let n be a node (which is neither `head` nor `tail`), and let b be the initial value of its two validity bits.

(1) We say that n is at its *initial state* if the value of both of its validity bits is b.
(2) We say that n is *invalid* if the value of its first validity bit is $\neg b$ and the value of its second validity bit is b.
(3) We say that n is *valid* if the value of both of its validity bits is $\neg b$.
(4) We say the n is *marked* if its *next* pointer is marked. Otherwise, we say that n is *unmarked.*

The `head` and `tail` sentinel nodes are always considered as valid and unmarked nodes.

We now prove some basic claims regarding the link-free list implementation.

Claim B.4 (State Transitions). *Let n be a volatile node. Then its state can only go through the following transitions:*

(1) *From being unmarked and in its initial stage, to being unmarked and invalid.*
(2) *From being unmarked and invalid, to being unmarked and valid.*
(3) *From being unmarked and valid, to being marked and valid.*

Proof. A node n is always created with an initialized and unmarked state, and its state can only change in line 11 of Listing 3, line 6, 12 or 18 of Listing 4, or in line 10 or 11 of listing 5. As explained is Section 3.1, executing the `flipV1` or `makeValid` auxiliary functions on the same node more than once, would not effect its state. Moreover, when `makeValid` is executed before `flipV1`, the node's state remains initialized and is also not effected. Therefore, `flipV1` only changes the node's state from being initialized to being invalid, `makeValid` only changes the node's state from being invalid to being valid, and it remains to show that the marking of a node does not foil the above transition types. Since a node can only be marked (line 11 of listing 5), and is never unmarked throughout the execution, we only need to show that it is valid when marked. If it is either invalid or valid before executing line 10, then from the above, it is valid when marked in line 11. Notice that it also cannot be at its initialized state, since a node becomes invalid right after its creation, in line 12 of Listing 4. □

Claim B.5 (Marked Nodes). *Once a node is marked, its* next *pointer does not change anymore.*

Proof. Let n be a marked node. From Claim B.4, it cannot be unmarked. Besides when marked in line 11 of listing 5, n's *next* pointer can only change during a successful CAS execution in line 4 of Listing 2 or line 17 of Listing 4. In both cases, it is assumed that n is unmarked and therefore, the CAS execution is unsuccessful if it is marked, leaving n's *next* pointer unchanged. □

Claim B.6 (The States of the Sentinel Nodes). *The* `head` *and* `tail` *sentinel nodes are always unmarked and valid.*

Proof. As mentioned in the proof of Claim B.4, a node's state can only change when its key is sent as an input parameter to one of the list's operations. Assuming the neither $-\infty$ nor ∞ are sent as input parameters to the list's operations, the states of the `head` and `tail` sentinel nodes always remain unmarked and valid. □

Claim B.7 (Nodes Invariants). *Let n_1 and n_2 be two different nodes. Then:*

(1) *If n_2 is the successor of n_1 in the list then n_2 is not an infant.*
(2) *Right before executing line 17 in Listing 4, having $newNode = n_2$, it holds that: (1) n_2 is an infant, and (2) n_2 is invalid.*
(3) *If n_2 is not an infant and not marked, or marked but not yet flushed since being marked, then n_2 is reachable.*
(4) *If n_2 is marked, but has not been flushed since being marked, then n_2 is reachable.*
(5) *If n_1's key is smaller than or equal to n_2's key, then n_1 is not reachable from n_2.*
(6) *If n_2 is reachable from n_1 at a certain point, then as long as n_2 is not marked, n_2 is still reachable from n_1.*
(7) *If n_1 is not an infant then the* `tail` *sentinel node is reachable from n_1.*

Proof. We are going to prove the claim by induction on the length of the execution. At the initial stage, `head` and `tail` are the only nodes in the list, having $-\infty$ and ∞ keys (respectively), both are reachable by Definition B.1, and `head` is `tail`'s predecessor. Therefore, all of the invariants obviously hold. Now, assume that all of the invariants hold at a certain point during the execution, at let s be the next execution step, executed by a thread t.

(1) If n_2 is not an infant before executing s, then by Definition B.2, it is not an infant after executing s, and the invariant holds. Otherwise, by the induction hypothesis, n_2 does not have a predecessor before executing s, and it cannot be the `head` sentinel node. n_1's successor can only change in line 4 of Listing 2, or in line 16 or 17 of Listing 4. If s is the execution of line 4 in Listing 2 or line 16 in Listing 4, then n_2 has already been traversed during a former `find` execution, as a node with a predecessor, and by the induction hypothesis, is not an infant. If s is the execution of line 17 in Listing 4, then n_2 is not an infant by Definition B.2.
(2) Since n_2 can only be that node during the execution of the `insert` operation in which it is created, and which returns in line 20, after a successful CAS execution in line 17, by Definition B.2, n_2 must be an infant at this point, and (1) holds. Now, assume by contradiction that n_2 is not invalid. Since it becomes invalid in line 12 and by Claim B.4, its state must be valid. n_2's state can become valid only in line 11 of Listing 3, in line 6 or 18 of Listing 4, or in line 10 of Listing 5. In all cases, it must have a predecessor prior to that change, and by invariant 1, it is not an infant – a contradiction. Therefore, n_2's state is invalid, and the invariant holds.
(3) If n_2 was an infant before executing s, then s is the execution of line 17 in Listing 4, making n_2 the successor of some node which is reachable by assumption. n_2 is reachable in this case. Otherwise, by assumption and Claim B.4, it was reachable right before executing s. Assume

by contradiction that it is no longer reachable after executing s. Then n_2 is reachable from a node n_1 that was reachable right before s, and is no longer reachable (may be n_2 itself). Assume w.l.o.g that n_1 is such a node for which the path of nodes from Definition B.1 is the longest. The node n_1 can only become unreachable if the current step is the execution of line 4 in Listing 2, and if n_1 is marked and then flushed in line 2 of Listing 2. This means that $n_1 \neq n_2$. Since n_1's successor stays reachable in this case, we get a contradiction. Therefore, n_2 is reachable in this case as well.

(4) By assumption, n_1 is not reachable from n_2 right before executing s. Since all changes of nodes' successors (line 4 of Listing 2, and line 16 and 17 of Listing 4) preserve keys order (notice the halting condition in line 11 of Listing 2), the Invariant still holds.

(5) If n_2 is not reachable from n_1 before executing s then the invariant holds vacuously. Otherwise, assume by contradiction that n_2 was reachable from n_1 right before executing s, and is no longer reachable from n_1 after executing it. Let n_3 be the first node reachable from n_1 after the previous step, that is not reachable from it after executing the current step (n_3 must exist). The node n_3 can only become unreachable from n_1 if the current step is the execution of line 4 in Listing 2, and if n_3 is marked. This means that $n_3 \neq n_2$. Since n_3's successor stays reachable from n_1 in this case, we get a contradiction. Therefore, n_2 is still reachable from n_1.

(6) If n_1 was an infant right before executing s then s is executing a successful CAS in line 17 of Listing 4. In this case, s makes n_1 the predecessor of a node whose `tail` is reachable from, by assumption. Therefore, `tail` is reachable from n_1 in this case. Otherwise, assume by contradiction that `tail` was reachable from n_1 right before executing s (must hold by assumption), but is no longer reachable from it after executing it. Let n_2 be the last node reachable from n_1, for whom `tail` is not reachable from after executing the current step (n_2 must exist). Then the current step must change n_2's *next* pointer. Since n_2 cannot be an infant (by Invariant 1), this step is a successful CAS, either in line 4 of Listing 2 or in line 17 of Listing 4. In both cases, n_2's successor is set to be a node that `tail` is reachable from, by assumption. Since we get a contradiction to Definition B.1, `tail` is reachable from n_1 in this case as well.

□

Claim B.8 (The Volatile List Invariant). *The list is always sorted by the nodes' keys, no key ever appears twice, and the* `head` *and* `tail` *sentinel nodes are always the first and last members of the list, respectively.*

Proof. From Invariant 5 of Claim B.7, the volatile list is always sorted by the nodes' keys and no key ever appears twice. By Claim B.6 and Invariant 3 of Claim B.7, the `head` and `tail` sentinel nodes are always members of the list, and by Invariant 5 of Claim B.7, they are the first and last members, respectively. □

We now move to dealing with the persistent list. The persistent list contains the persistent copies of the volatile list's nodes, as long as their state is valid and not marked, as stated in Definition B.9 below.

Definition B.9 (Persistently in the List). Let n be a node. We say that n is *persistently in the list* if the state of n's persistent copy is valid and not marked.

Claim B.10 below asserts that being valid and not marked is sufficient for staying persistently in the list. In particular, the `head` and `tail` sentinel nodes always remain persistently in the list.

Claim B.10 (Being Persistently in the List). *Let n be a node which is persistently in the list. As long as n's state is valid and unmarked, n is still persistently in the list.*

Proof. Assume that n is persistently in the list at some point, and let assume by contradiction that there exists a later point, in which n's state is valid and unmarked, and is not persistently in the list. We are going to consider the earliest such point. By Claim B.4, n does not change between the mentioned two points. Therefore, each flush of n, flushes it with a valid and unmarked – a contradiction. Thus, n is still persistently in the list. □

Claim B.11 (Persistently in the List Nodes are Reachable). *Let n be a node which is persistently in the list. Then n is reachable.*

Proof. Assume that n is persistently in the list. By Definition B.9, during the last flush of n to the non-volatile memory, n's state was valid and unmarked. If n is unmarked, then by Invariants 2 and 3 of Claim B.7, n is reachable. Otherwise, since n is marked but still persistently in the list, n has not been flushed in line 2 of Listing 2, line 8 of Listing 3, or implicitly flushed yet, and in particular, it has not become unreachable in line 4 of Listing 2 yet (according to the proof of Claim B.7, it cannot become unreachable in other scenarios). Therefore, n is still reachable in this case as well. □

Notice that Claim B.11 does not hold temporarily during recovery, until the list is reconstructed. However, this fact does not effect the use of this claim throughout our proof.

Claim B.12 (The Persistent List is a Set). *The persistent list never contains two different persistent nodes with the same key.*

Proof. The claim derives directly from Claim B.8 and B.11. □

Claim B.13. *Let n_1 and n_2 be the two volatile nodes returned as output from the find method. Then during the method execution, there exist a point in which (1) n_1 is reachable, (2) n_2 is n_1's successor, and (3) n_2 is unmarked.*

Proof. When n_1's marked bit is read for the first time during the execution, it is unmarked (otherwise, it would have been trimmed and not returned). In addition, since it must have had a predecessor at an earlier point (otherwise, it would not have been traversed), from Invariant 1 of Claim B.7, it is not an infant, and from Invariant 3 of Claim B.7, it is reachable at this point. If n_2 is n_1 successor at this point, then the claim holds for this point. Notice that n_2 cannot be marked at this point, since otherwise, it would have been trimmed at a later point and not returned as output. If n_2 is not n_1's successor at this point, then there exists a point between the first read of n_1 and the first read of n_2 in which n_2 becomes n_1's successor. From Claim B.5, n_1 is unmarked at this point and thus, from Invariant 3 of Claim B.7, it is reachable at this point. In addition, n_2 is unmarked at this point as well, and the claim holds in this case. □

Claim B.14. *Let there be an insert execution that returns false in line 8 (Listing 4), and let m be the node returned as the second output parameter from the last find call in line 4. Then at least one of the following holds during the insert execution:*

(1) *m is persistently in the list.*
(2) *m is marked and then flushed.*

Proof. Claim B.13 guarantees that there exists a point during the last find execution in which m is reachable and unmarked. Since m is made valid no later than the execution of line 6, and is flushed, while being still valid (by Claim B.4), no later than the execution of line 7, it is either persistently in the list (by Definition B.9), or becomes marked before its flush. In both cases, the claim holds. □

Claim B.15. *Let n_2 be a node which is assigned into the* `curr` *variable in line 4 of Listing 3, and let n_1 be the last node assigned into the* `curr` *variable before n_2. Then there exists a point during the traversal in which both nodes are reachable and n_2 is n_1's successor.*

Proof. Assume by contradiction that the claim does not hold. W.l.o.g., Let n_1 and n_2 be the first two nodes for which (1) n_1 and n_2 are assigned into the `curr` variable sequentially, and (2) the guaranteed point does not exist for them. Since this point does exist for n_1 and the former node assigned into `curr`, n_1 is reachable at some point during the execution (if n_1 is the `head` sentinel node then it is obviously reachable). From Invariant 6 of Claim B.7, n_1 is reachable as long as it is not marked. Since n_2 is its successor when assigned into the `curr` variable, from Claim B.5 it was its successor at the last step in which n_1 was reachable before this assignment (might be the assignment itself). Therefore, there exists such a point for n_1 and n_2 – a contradiction, and the claim holds. ☐

B.1 Durable Linearizability

We use the notion of *durable linearizability* [Izraelevitz et al. 2016] for correctness. The recovery procedure, executed after a crash (and described in Section 3.5), is assumed to terminate before new threads start executing their code. Given an operation for which a crash event occurs after its invocation and before its response, we consider its response point as the end of the respective recovery procedure. Notice that in the following definitions, we do not consider recoveries that are interrupted by crash events. We do so for clarity and brevity. The definitions can be easily extended to include such cases.

We are going to prove that, given an execution, removing all crash events would leave us with a linearizable history, including all the operations that were fully executed between two crashes, and some of the operations that were halted due to crashes (and then recovered during recovery). We are going to define, per operation execution, whether it is *a surviving operation.* A surviving operation is an operation that is linearized in the final crash-free history of the execution (by removing all crash events). Obviously, operations that were fully executed between two crash events are always considered as surviving operations. Additionally, we are going to define the linearization points of all surviving operations in the crash-free history.

For each linearized operation, we define its linearization point as a point during its execution in which it takes effect. For a more accurate definition, we first define, in Definition B.16 below, which nodes are considered as set members. Given this definition, a successful insertion takes effect when a respective new node becomes a set member, a successful removal takes effect when an existing respective set member is removed from the set, a contains execution returns an answer which respects the set membership definition, and unsuccessful operations fail according to this definition as well.

Definition B.16 (Being a Set Member). Given a node n, it is considered as a set member as long as at least one of the following holds:

(1) n is persistently in the list according to Definition B.9.
(2) If n is marked and then flushed, for the first time since it becomes valid, then n is considered as a set member during the period in which it is valid and not yet flushed.

Notice that being persistently in the list is not effected by crash events (since it depends on the state saved in the non-volatile memory). Moreover, a node which is considered as a set member of the second type, stops being a set member before the next crash event. Therefore, being a set member is well-defined, even in the presence of crash events. For using the term of set membership in our durable linearizability proof, we still need to prove that the collection of all set members is indeed a set. We do so in Claim B.17.

Claim B.17. *Let n_1 and n_2 be two different set members. Then n_1's key is different from n_2's key.*

Proof. Assume by contradiction that n_1 and n_2 are two different set members with the same key. By Claim B.12, the persistent list never contains two different persistent nodes with the same key, and therefore, at least one of them is not persistently in the list. Assume, w.l.o.g., that n_1 is not persistently in the list.

Since n_1 is a set member, by Definition B.16, it is valid, and either not marked, or marked and not flushed yet. By Invariant 3 of Claim B.7, n_1 is reachable. By Claim B.8, there cannot be two reachable nodes with the same key, and therefore, n_2 is not reachable, and by Invariant 3 of Claim B.7, it is either not valid, or marked and flushed. In both cases, it is not a set member according to Definition B.16 – a contradiction. Therefore, there cannot exist two different set members with the same key, and the claim follows. □

We are now going to define, per operation, the terms for being considered as a surviving operation (in the presence of a crash event), its respective linearization point. In addition, we are going to prove that each survivng operation indeed takes effect at its linearization point, and that non-surviving operations do not take effect at all.

B.1.1 Insert. Before defining the conditions for the survival of an insert operation, we need to re-define the success of an insertion in the presence of crash events.

Definition B.18 (A Successful Insert Operation). Given an execution of an insert operation, we say that this operation is successful if one of the following holds before any crash event, following its invocation:

(1) The operation returns `true`.
(2) A node n is allocated in line 11, becomes valid, and is flushed afterwards (not necessarily in the scope of the operation in which it is allocated).

The operation is unsuccessful if it returns `false`.

Definition B.19 (A Surviving Insert Operation). An insert operation is considered as a surviving operation if, before the first crash event that follows its invocation, one of the following holds:

(1) The operation is unsuccessful according to Definition B.18. Let m be the node returned as the second output parameter from the last find call in line 4. The operation's linearization point is set to be a point, during the execution, in which m is a set member according to Definition B.16 (chosen arbitrarily).
(2) The operation is successful according to Definition B.18, and the node allocated in line 11 becomes persistently in the list (see Definition B.9) before the crash event. In this case, the linearization point is set to be the flush which inserts it to the persistent list.
(3) The operation is successful according to Definition B.18, and the node allocated in line 11 does not become persistently in the list before the first crash event. In this case, the linearization point is set to be the step which changes its state to valid.

Claim B.20. *A surviving insert operation takes effect instantaneously at its linearization point.*

Proof. We are going to prove the claim for each of the three surviving insertion types.

(1) Suppose that the operation is unsuccessful according to Definition B.18, and let m be the node returned as the second output parameter from the last find call in line 4. Notice that m's key is equal to the key received as input. We are going to show that m is a set member

at the linearization point defined in Definition B.19, and therefore, the (unsuccessful) insert operation indeed takes effect at this point. According to Claim B.14, there must exist a point during the execution in which *m* is either persistently in the list, or that it is marked and then flushed. In the first scenario, by Definition B.16, *m* is indeed a set member, and we are done. In the second scenario, it is guaranteed by Claim B.13 that *m* is marked during the execution (since it is valid and unmarked at some point during the find method execution). Therefore, a point at which it is a set member, exists according to Definition B.16, and the claim holds.

(2) Suppose that the operation is successful according to Definition B.18, and the node allocated in line 11 becomes persistently in the list (see Definition B.9) before the crash event. By Definition B.16, the allocated node indeed becomes a set member at the linearization point defined above and thus, the operation takes effect instantaneously at this point.

(3) Suppose that the operation is successful according to Definition B.18, and the node allocated in line 11 does not become persistently in the list before the first crash event. By Definition B.18, it becomes valid, then marked, and then flushed, during the execution, and before any crash event. By Definition B.16, it becomes a set member when its state becomes valid and thus, the operation indeed takes effect at its defined linearization point.

□

Claim B.21. *A non-surviving insert operation takes no effect.*

Proof. By Definition B.19, during a none-surviving insert operation, if a volatile node is allocated, and even if it is inserted into the volatile list, and becomes valid, it is not flushed. By Definition B.16, it is not considered as a set member. In particular, it is not persistently in the list and thus, will also not be considered as a set member after a crash event. □

B.1.2 Remove. We also re-define the success of a removal in the presence of crash events.

Definition B.22 (A Successful Remove Operation). Given an execution of a remove operation, we say that this operation is successful if one of the following holds before any crash event, following its invocation:

(1) The operation returns `true`.
(2) A node *n* is marked in line 11 and is flushed afterwards (not necessarily in the scope of the operation in which it is marked).

The operation is unsuccessful if it returns `false`.

Definition B.23 (A Surviving Remove Operation). A remove operation is considered as a surviving operation if, before the first crash event that follows its invocation, one of the following holds:

(1) The operation is unsuccessful according to Definition B.22. The operation's linearization point is set to be the point guaranteed by Claim B.13.
(2) The operation is successful according to Definition B.22. The operation's linearization point is set to be the first flush of the victim node, after its marking in line 11.

Claim B.24. *A surviving remove operation takes effect instantaneously at its linearization point.*

Proof. We are going to prove the claim for each of the two surviving removal types.

(1) Suppose that the operation is unsuccessful according to Definition B.22, and let *m* be the node returned as the second output parameter from the last find call in line 5. Notice that *m*'s key is different from the key received as input. By Claim B.13, *m* is reachable at the linearization point. Moreover, its key is bigger than the key received as input, its predecessor's key is smaller than this key (by the find specification) and by Claim B.8, there does not exist a

reachable node with the input key. Since being a set member implies being reachable, there does not exist a set member with the given key at its linearization point and thus, it indeed takes effect this point.

(2) Suppose that the operation is successful according to Definition B.22, and the node marked in line 11 is flushed afterwards, and before the following crash event. If the non-volatile memory already contains a valid and unmarked copy of this node, then the operation's linearization point (according to Definition B.23) indeed removes this node from the set, according to Definition B.16. Otherwise, the mentioned flush is the first flush of the victim node, and according to Definition B.16, it removes it from the set in this case as well.

□

Claim B.25. *A non-surviving remove operation takes no effect.*

Proof. By Definition B.23, during a none-surviving remove operation, if a victim node is found, and even if it is made valid and marked, it is not flushed. By Definition B.16, whether it is originally a set member or not, it is not removed from the set. □

B.1.3 Contains. We do not use the term of success for describing a contains execution, and, therefore, the terms for its survival are straight forward.

Definition B.26 (A Surviving Contains Operation). A contains operation is considered as a surviving operation if it terminates before the first crash event that follows its invocation. For defining linearization points per contains execution, let n_1 and n_2 be the last nodes assigned into the `curr` variable.

(1) When the operation returns `true`, its linearization point is set to be a point during the execution in which n_2 is a set member (chosen arbitrarily).
(2) When the operation returns `false` in line 6, its linearization point is set to be the point guaranteed by Claim B.15 for n_1 and n_2.
(3) When the operation returns `false` in line 9, its linearization point is set to be a point during the execution in which n_2 is reachable but not a set member (chosen arbitrarily).

Claim B.27. *A surviving contains operation takes effect instantaneously at its linearization point.*

Proof. Let n_1 and n_2 be the last nodes assigned into the `curr` variable. We are going to prove the claim for each of the three surviving contains types.

(1) Suppose that the operation returns `true`. We are going to show that there indeed exists a point during the execution in which n_2 is a set member. Since the operation does not return in line 9, from Claim B.4, n_2 is not marked during the traversal. In addition, since n_2 is made valid at the latest when executing line 11, from Claim B.4, it is also valid when executing line 12. There are several possible scenarios:
   (a) n_2 is not marked during the execution. In this case, n_2 becomes a set member at the latest when executing line 12. In this case, there obviously exists a point during the execution at which n_2 is a set member.
   (b) n_2 is marked during the execution, and is flushed at some point after becoming valid and before becoming marked (by Claim B.4, n_2 becomes valid before it is marked). There exists a suitable point in this case as well.
   (c) The remaining case is when n_2 is not flushed after becoming valid and before being marked. In this case, it is flushed at the latest in line 12, and therefore, by Definition B.16, it is a set member at some point, before being marked.

   There exists a suitable linearization point in every case.

(2) Suppose that the operation returns `false` in line 6. Claim B.15 guarantees that both n_1 and n_2 are reachable at this point. Since n_1's key must be smaller then the key received as input, and n_2 must be bigger, by Claim B.8, there does not exist a reachable node with the given key at this point. By Claim B.11, there does not exist a set member with the given key at this point.
(3) Suppose that the operation returns `false` in line 9. If it still reachable when executing line 8, then a marked copy of n_2 resides in the non-volatile memory (i.e., it is not a set member by Definition B.16), while n_2 is still reachable, and the guaranteed point exists. Otherwise, before it becomes unreachable (which happens during the contains execution, according to Claim B.15), at the latest, it is flushed as a marked node in line 2 of Listing 2. Therefore, the guaranteed point exists in this case as well. By Claim B.8 and B.11, there does not exist a set member with the given key at this point.

□

Since a contains operation does not effect the list (it executes flushes, that can also be executed implicitly), there is no need to prove that non-surviving contains executions do not take effect.

Theorem B.28. *The link-free list is durable linearizable.*

Proof. By Definition B.19, B.23 and B.26, all the operations that are fully executed between two crashes (and some of the operations that are halted due to crash events), have a linearization point. By Claim B.20, B.24 and B.27, each operation takes effect instantaneously at its linearization point. By Claim B.21 and B.25, operations for which we did not define linearization points (non-surviving operations), do not take effect at all. In summary, the link-free list is durable linearizable by definition [Izraelevitz et al. 2016]. □

B.2 Lock-Freedom

B.2.1 A Preliminary Discussion. Lock-freedom is impossible to show in the presence of crashes. To see that this is the case, imagine an adversarial schedule of crashes that repeatedly creates a crash one step before the completion of an operation. Such crashes can also occur during the recovery process itself. As far as we know, lock-freedom has not been previously discussed in the presence of crashes.

One way to deal with this problem is to admit that in the presence of crashes lock-freedom cannot be guaranteed, but as crashes are expected to occur infrequently, this still leaves the question of lock-freedom during crash-free executions. Such lock-freedom is of high value in practice, when crashes are indeed rare. A more theoretical approach is to consider crashes as progress, as if a crash itself is one of the operations on the data structure. Interestingly, this yields the same challenge. While executions with crashes always make progress, crash-free executions need a proof of progress. So in what follows we prove that the link-free list is lock-free in the absence of crashes.

We are going to prove that in crash-free executions, at least one of the operations terminates. To derive a contradiction, assume there is some execution for which no executing operation terminates after a certain point. Notice that we can assume that no operation is invoked after this point, and that the set of running operations is finite (since there is a finite number of system threads). The rest of the proof relates to the suffix α of the execution, starting from this point.

Claim B.29. *There is a finite number of state changes of reachable nodes during α.*

Proof. A contains execution must terminate after executing line 11, an insert execution must terminate after executing line 6 or 18, and a remove execution must terminate after a successful CAS execution in line 11. In addition, the state change in line 12, during an insert execution, is of an unreachable node. Consequently, we can assume that after a certain point, state changes are made only in line 10, of Listing 5. Since a finite number of new nodes is created and made reachable during α (at most one node per pending insert operation), and since every such node eventually becomes valid in line 18 of Listing 4, we can assume that the number of state changes in line 10 of Listing 5 is finite as well. □

Claim B.30. *There is a finite number of pointer changes of reachable nodes during α.*

Proof. The pointers of reachable nodes change either in line 4 of Listing 2 or line 17 of Listing 4. A state change in line 17 of Listing 4 would cause the termination of an insert execution and thus, the only pointer changes are physical removals of marked nodes, executed in line 4 of Listing 2. Since there is a finite number of state changes of reachable nodes during α (by Claim B.29), the number of marked nodes is bounded and thus, there is a finite number of pointer changes of reachable nodes during α. □

Theorem B.31. *The link-free list is lock-free.*

Proof. From Claims B.29 and B.30, after a certain point, there are no state or pointer changes in the list. Therefore, we consider the suffix α' of the execution that contains no state or pointer changes of reachable nodes. Obviously, starting from this point, the list becomes stable, and does not change anymore.

Since the list is finite, from Claim B.8, every find and contains execution eventually ends. In addition, every insert and remove operation must be unsuccessful, and also terminate (since calls to the find method always terminate). We get a contradiction and therefore, the implementation is lock-free. □

C SOFT CORRECTNESS

In this section we prove the correctness (i.e., durable linearizability) and progress guarantee (lock-freedom) of the soft list. We start by proving some volatile list invariants. In Section C.1 we prove the linearizability of our implementation when there are no crash events, followed by a durable linearizability proof in Section C.2. Finally, we show our implementation is lock-free in Section C.3.

Claim C.1 (State Transitions). *The state of a volatile node can only go through the following transitions:*

(1) *From "intend to insert" to "inserted"*
(2) *From "inserted" to "inserted with intention to delete"*
(3) *From "inserted with intention to delete" to "deleted"*

Proof. A node's state can change either in line 33 of Listing 11, or in line 14 or 17 of Listing 12. In all three cases, the state changes according to one of the options mentioned above, and the claim follows immediately. Notice that in the rest of the assignments into a node's *next* pointer (line 5 of Listing 9 and line 23 of Listing 11), the state stays unchanged. □

Claim C.2 (Deleted States). *Once the state of a node becomes "deleted", its* next *pointer does not change anymore.*

Proof. A node's *next* pointer changes either in line 5 of Listing 9 or in line 23 of Listing 11. In both cases, the state of the node whose *next* pointer is to be updated, is checked before the update (guaranteeing that its state is not "deleted"), and the CAS execution ensures that it does not change until the pointer changes (from Claim C.1, its state cannot become "deleted" and change again afterwards). Notice that we deal with state changes in Claim C.1. In this claim we refer only to reference changes. □

Claim C.3 (The States of the Sentinel Nodes). *The states of the* head *and* tail *sentinel nodes are always "inserted".*

Proof. As mentioned in the proof of Claim C.1, a node's state can change either in line 33 of Listing 11, or in line 14 or 17 of Listing 12. In all three cases, the node's key is sent as an input parameter to the insert or remove operation, respectively. Assuming the neither $-\infty$ nor ∞ are sent as input parameters to the insert and remove operations, the states of the head and tail sentinel nodes always remain "inserted". □

Definition C.4 (Reachability). We say that a volatile node n is *reachable* from a volatile node n' if there exists nodes $n_0, n_1, \ldots, n_k$ such that $n_0 = n'$, $n_k = n$ and for every $0 \le i < k$, n_i is the predecessor of n_{i+1} in the list. We say that a node n is *reachable* if it is reachable from the head sentinel node.

Definition C.5 (Logically in the List). We say that a volatile node n is logically in the list if n is reachable and its state is either "inserted" or "inserted with intention to delete".

Definition C.6 (Infant Nodes). We say that a volatile node n is an *infant* if n is neither head nor tail, and there does not exist an earlier successful execution of the CAS operation in line 23 in Listing 11, satisfying $newNode = n$.

Claim C.7 (Volatile Nodes Invariants). *Let n_1 and n_2 be two different volatile nodes. Then:*

(1) *If n_2 is the successor of n_1 then n_2 is not an infant.*
(2) *Right before executing line 23 in Listing 11, having $newNode = n_2$, it holds that: (1) n_2 is an infant, and (2) n_2's state is "intend to insert".*
(3) *If n_2 is not an infant and its state is not "deleted", then n_2 is reachable.*
(4) *If n_1's key is smaller than or equal to n_2's key, then n_1 is not reachable from n_2.*
(5) *If n_2 is reachable from n_1 at a certain point, then as long as n_2's state is not "deleted", n_2 is still reachable from n_1.*
(6) *If n_1 is not an infant then the* tail *sentinel node is reachable from n_1.*

Proof. In the initial stage, the head and tail sentinels are the only volatile nodes in the list, both with an "inserted" state, and tail is head's successor. Invariant 1 holds since tail is not an infant, Invariant 2 holds vacuously, Invariants 3, 5 and 6 hold since both head and tail are reachable, and Invariant 4 holds since head is not reachable from tail.

Now, assume all invariants hold until a certain point during the execution. We are going to prove that they also hold after executing the next step by one of the system threads.

(1) If n_2 was also n_1's successor before the current step, then by assumption, it is not an infant. Otherwise, n_1's *next* pointer was updated to point to n_2 in the current step, either in line 5 of Listing 9, in line 20 of Listing 11, or in line 23 of Listing 11. In the first two cases, there exists an earlier point during the execution, in which n_2 is the successor of a certain node (during the execution of the find method). By assumption, n_2 is not an infant in these cases. In the third case, after executing the current step, n_2 is not an infant by Definition C.6.
(2) Assume that the next step will execute line 23 of Listing 11, having $newNode = n_2$. Assume by contradiction that n_2 is not an infant. Since the CAS in line 23 can only be executed on nodes created in line 18, by the creating thread, n_2 is an infant and (1) holds. Now, assume by contradiction that n_2's state is not "intend to insert". Then it had been changed in line 33 of Listing 11, during another insert execution, implying that, by Invariant 1 and the choice of the resultNode variable, n_2 is the successor of some node and thus, is not an infant – a contradiction. Therefore, n_2's state is "intend to insert" and (2) holds as well.

(3) If n_2 was an infant before the current step, then the current step is the execution of line 23 in Listing 11, making n_2 the successor of some node which is reachable by assumption. n_2 is reachable in this case. Otherwise, by assumption and Claim C.1, it was reachable during the former step. Assume by contradiction that it is no longer reachable after executing the current step. Then n_2 is reachable from a node n_1 that was reachable after the previous step, and is no longer reachable (may be n_2 itself). Assume w.l.o.g that n_1 is such a node for which the path of nodes from Definition C.4 is the longest. The node n_1 can only become unreachable if the current step is the execution of line 5 in Listing 9, and if n_1's state is "deleted". This means that $n_1 \neq n_2$. Since n_1's successor stays reachable in this case, we get a contradiction. Therefore, n_2 is reachable in this case as well.

(4) By assumption, n_1 is not reachable from n_2 after the previous step. Since all changes of nodes' successors (line 5 in Listing 9 and lines 20 and 23 in Listing 11) preserve keys order (notice the halting condition in line 17 of Listing 9), the Invariant still holds.

(5) If n_2 is not reachable from n_1 after the previous step then the invariant holds vacuously. Otherwise, assume by contradiction that n_2 was reachable from n_1 after the previous step, and is no longer reachable from n_1 after the current step. Let n_3 be the first node reachable from n_1 after the previous step, that is not reachable from it after executing the current step (n_3 must exist). The node n_3 can only become unreachable from n_1 if the current step is the execution of line 5 in Listing 9, and if n_3's state is "deleted". This means that $n_3 \neq n_2$. Since n_3's successor stays reachable from n_1 in this case, we get a contradiction. Therefore, n_2 is still reachable from n_1.

(6) If n_1 was an infant after the previous step then the current step (executing a successful CAS in line 23 of Listing 11) makes n_1 the predecessor of a node whose `tail` is reachable from, by assumption. Therefore, `tail` is reachable from n_1 in this case. Otherwise, assume by contradiction that `tail` was reachable from n_1 after the previous step (must hold by assumption), but is no longer reachable from it after the current step. Let n_2 be the last node reachable from n_1, for whom `tail` is not reachable from after executing the current step (n_2 must exist). Then the current step must change n_2's *next* pointer. Since n_2 cannot be an infant (by Invariant 1), this step is a successful CAS, either in line 5 of Listing 9 or in line 23 of Listing 11. In both cases, n_2's successor is set to be a node that `tail` is reachable from, by assumption. Since we get a contradiction to Definition C.4, `tail` is reachable from n_1 in this case as well.

□

Claim C.8 (The Volatile List Invariant). *The volatile list is always sorted by the nodes' keys, no key ever appears twice, and the* `head` *and* `tail` *sentinel nodes are always the first and last members of the list, respectively.*

Proof. From Invariant 4 of Claim C.7, the volatile list is always sorted by the nodes' keys and no key ever appears twice. By Claim C.3 and Invariant 3 of Claim C.7, the `head` and `tail` sentinel nodes are always members of the list, and by Invariant 4 of Claim C.7, they are the first and last members, respectively. □

Claim C.9 (Being Logically in the Volatile List). *A volatile node n is logically in the list if and only if its state is either "inserted" or "inserted with intention to delete".*

Proof. By Definition C.5, if n is logically in the list then its state is either "inserted" or "inserted with intention to delete". It remains to show that if its state is either "inserted" or "inserted with intention to delete" then it is reachable and, thus, logically in the list by Definition C.5. When n's state was changed from "intend to insert" to "inserted" in line 33 of Listing 11, it must have had a predecessor. From Invariant 1 of Claim C.7, it is not an infant. From Invariant 3 of Claim C.7, it is reachable. □

C.1 Linearizability

We define linearization points for the insert, remove and contains operations, as well as for the find auxiliary method. We explicitly specify the linearization points of the linked-list when no crashes occur.

C.1.1 Find. We define the linearization point of the find method to be the point guaranteed from Claim C.10 below.

Claim C.10. *Let n_1 and n_2 be the two volatile nodes returned as output from the find method. Then during the method execution, there exist a point in which (1) n_1 is reachable, (2) n_2 is n_1's successor, and (3) n_2's state is not "deleted".*

Proof. When n_1's state is read for the first time during the execution, it is not "deleted" (otherwise, it would have been trimmed and not returned). In addition, since it must have had a predecessor at an earlier point (otherwise, it would not have been traversed), from Invariant 1 of Claim C.7, it is not an infant, and from Invariant 3 of Claim C.7, it is reachable at this point. If n_2 is n_1 successor at this point, then the claim holds for this point. Notice that n_2's state cannot be "deleted" at this point, since otherwise, it would have been trimmed at a later point and not returned as output. If n_2 is not n_1's successor at this point, then there exists a point between the first read of n_1 and the first read of n_2 in which n_2 becomes n_1's successor. From Claim C.2, n_1's state is not "deleted" at this point and thus, from Invariant 3 of Claim C.7, it is reachable at this point. In addition, n_2's state is not "deleted" at this point as well, and the claim holds in this case. □

C.1.2 Insert. Let n be the volatile node created during a successful execution of the insert operation (line 18 in Listing 11). Since the operation returns *true*, it is guaranteed that n's state changes from "intend to insert" to "inserted" in line 33. We define the linearization point of a successful insert operation at this point. From Claim C.1 and C.9, this is indeed the first point during the execution in which n is logically in the list.

Now, let there be an unsuccessful execution of the insert operation, and let m be the volatile node returned as the second output parameter from the find call in line 6. Since the condition checked in line 10 must hold, its key is equal to the key received as input. Claim C.11 below guarantees that during the execution there exists a point in which m is logically in the list. We set this point as the operation's linearization point in this case.

Claim C.11. *There exists a point between the linearization point of the mentioned find execution and the return of the operation in which m's state is either "inserted" or "inserted with intention to delete".*

Proof. If m's state, read in line 11, is not "intend to insert", then From Claim C.10 it is guaranteed that at the linearization point of the find execution, m's state is not "deleted". If it is either "inserted" or "inserted with intention to delete", then from Definition C.5, we are done. Otherwise, it is "intend to insert". However, when checking its state in line 11, it is not "intend to insert" (since the operation is unsuccessful, and the condition checked in line 11 must hold). From Claim C.1, it is guaranteed that before checking this condition, there exists a point in which m's state became "inserted", and the claim holds.

The remaining case is when the state read in line 11 is "intend to insert". In this case, the executing thread does not return before m's state changes (the condition checked in line 32 holds). From

Claim C.1, it is guaranteed that there exists a point in which m's state is "inserted", and the claim holds in the case as well. □

C.1.3 Remove. Let n be the volatile node returned from the find method call in line 5 of Listing 12.

If the operation returned in line 9 then its linearization point is defined at the linearization point of the find call from line 5. The find call returned two nodes that, from Claim C.10, are guaranteed to be reachable and successive at its linearization point. From Claim C.8 it is guaranteed that there does not exist a reachable node with the given key, and in particular, there does not exist a node with the given key which is logically in the list at this point.

If the operation returned in line 11, then the linearization point is the read of `currState` during the find execution. Since it was returned from the find call, from Invariant 1 of Claim C.7, it is not an infant. In addition, since its state is "intend to insert", from Invariant 3 of Claim C.7, it is reachable. By Definition C.5, it is not logically in the list, and by Claim C.8, there does not exist another node with the given key, which is reachable and in particular, logically in the list at this point.

Otherwise, the operation returned in line 21. It is guaranteed from Claim C.10 that at the linearization point of the find call, n's state was not "deleted". Since the loops in lines 13–14 and 16–17 terminated before the return from the operation in line 21, from Claim C.1, n's state was changed from "inserted with intention to delete" to "deleted" at some point between the linearization point of the find method and the return from the operation. This is the operation's linearization point in this case. From Claim C.9, it is guaranteed that the node stopped being logically in the list exactly at this step.

C.1.4 Contains. Let n be the last volatile node assigned into the `curr` variable in line 12 of Listing 10.

Claim C.12. *Let m be the last node assigned into the* `curr` *variable before n. Then there exists a point during the traversal in which both nodes are reachable and n is m's successor.*

Proof. Assume by contradiction that the claim does not hold. Let n_1 and n_2 be the first two nodes for which (1) n_1 and n_2 are assigned into the `curr` variable sequentially, and (2) the guaranteed point does not exist for them. Since this point does exist for n_1 and the former node assigned into `curr`, n_1 is reachable at some point during the execution. From Invariant 5 of Claim C.7, n_1 is reachable as long as its state is not "deleted". Since n_2 is its successor when assigned into the `curr` variable, from Claim C.2 it was its successor at the last step in which n_1 was reachable before this assignment (might be the assignment itself). Therefore, there exists such a point for n_1 and n_2 – a contradiction, and the claim holds. □

If n's key is not equal to the key received as input, then the linearization point is set to be the point guaranteed from Claim C.12. From Claim C.8, it is guaranteed that there does not exist a reachable node with the given key at this point.

Otherwise, n's key is equal to the key received as input. If its state, when executing line 13, is either "inserted" or "inserted with intention to delete", then the operation's linearization point is the read of its state in line 13. From Claim C.9, n is logically in the list at this point.

If its state is "intend to insert" when executing line 13, then the linearization point is set to be the one guaranteed from Claim C.12, in which n is reachable. From Claim C.1, n's state at this point is "intend to insert" as well and, thus, it is not logically in the list. From Claim C.8, since n is reachable, there does not exist another reachable (and in particular, which is logically in the list) node with the given key at this point.

If *n*'s state is "deleted" when executing line 13 and its state at the point guaranteed from Claim C.12 is also "deleted", then this point is the operation's linearization point. From the above reasons, there does not exist a node with the given key which is logically in the list at this point.

The remaining case is when *n*'s state is not "deleted" at the point guaranteed from Claim C.12, but it is "deleted" when executing line 13. Since its state is eventually "deleted", there exists a point between the guaranteed point and the execution of line 13 in which *n* state was changed to "deleted" and this is the operation's linearization point in this case. From Invariant 5 of Claim C.7, *n* is reachable at this point and therefore, from the above reasons, there does not exist a node with the given key which is logically in the list at this point in this case as well.

C.2 Durable Linearizability

As in Section B.1, we use the notion of *durable linearizability* [Izraelevitz et al. 2016] for correctness. The recovery procedure, executed after a crash (and described in Section 4.6), is assumed to terminate before new threads start executing their code. Given an operation for which a crash event occurs after its invocation and before its response, we consider its response point as the end of the respective recovery procedure. Notice that in the following definitions, we do not consider recoveries that are interrupted by crash events. We do so for clarity and brevity. The definitions can be easily extended to include such cases.

Before diving into the durable linearizability proof, we prove some basic claims regarding the persistent nodes, used during recovery.

Claim C.13 (State Transitions of Persistent Nodes). *The state of a persistent node can only go through the following transitions:*

(1) *From* valid *and* removed *to* invalid
(2) *From* invalid *to* valid *and not* removed
(3) *From* valid *and not* removed *to* valid *and* removed

Proof. Let *p* be a persistent node, allocated in line 18 of Listing 11, and let v be the negation of its `validStart` bit, when allocated (i.e., v is assigned into the `pValidity` field of the respective volatile node). When *p* is allocated, its state is *valid* and *removed*. The state of *p* can only change when creating or destroying it (Listing 7). The `create` method can only be called from line 30 of Listing 11, and the `destroy` method can only be called from line 15 of Listing 12, both with v as their `pValidity` input parameter. Notice that the first `create` execution terminates before the first `destroy` invocation, since the state of the respective volatile node is set to "inserted" in line 33 of Listing 11, only after the termination of the first `create` call, and is set to "inserted with intention to delete" in line 14 of Listing 12, before the first invocation of the `destroy` method (and by Claim C.1, a volatile node's state can be "inserted" only before it becomes "inserted with intention to delete").

The first `create` execution changes *p*'s state to *invalid* and then *valid* and not *removed*. Any further `create` calls do not change its state at all (since the value of the `validStart` and `validEnd` bits is already v). Therefore, any `destroy` call can only change it from *valid* and not *removed* to *valid* and *removed* (since it only changes the `deleted` bit), and the claim follows. □

Claim C.14 (Non-Removed Persistent Nodes). *Let n be a volatile node, and assume its representing persistent node has already been created in line 30 of Listing 11. If n's state is either "intention to insert" or "inserted", then the state of its representing persistent node is* valid *and not* removed.

Proof. As shown in the proof of Claim C.13, the state of *n*'s representing persistent node becomes *valid* and not *removed* when it is created in line 30 of Listing 11. In addition, from Claim C.13, it can only become *valid* and *removed*, after *n*'s state becomes "inserted with intention to delete". Since *n*'s state is either "intention to insert" or "inserted", the state of its representing persistent node remains *valid* and not *removed*. □

Claim C.15 (Removed Persistent Nodes). *Let n be a volatile node, and assume its representing persistent node has already been marked as* removed *in line 15 of Listing 12. Then the state of its representing persistent node does not become* valid *and not* removed *anymore.*

Proof. As shown in Claim C.13, any further `create` or `destroy` calls would not effect the persistent node's state. □

We are going to prove that, given an execution, removing all crash events would leave us with a linearizable history, including all the operations that were fully executed between two crashes, and some of the operations that were halted due to crashes (and then recovered during recovery). We are going to define, per operation execution, whether it is *a surviving operation.* A surviving operation is an operation that is linearized in the final crash-free history of the execution (by removing all crash events). Obviously, operations that were fully executed between two crash events are always considered as surviving operations. Additionally, we are going to define the linearization points of all surviving operations in the crash-free history.

C.2.1 Insert. Before defining the conditions for the survival of an insert operation, we need to re-define the success of an insertion in the presence of crash events.

Definition C.16 (A Successful Insert Operation). Given an execution of an insert operation, we say that this operation is successful if one of the following holds:

(1) The operation returns `true`.
(2) A volatile node *n* is allocated in line 18, the `result` variable is assigned with `true` in line 26, and the respective persistent node of *n* is created in line 30 by some thread before any crash event.

The operation is unsuccessful if it returns `false`.

Definition C.17 (A Surviving Insert Operation). An insert operation is considered as a surviving operation if, before the first crash event that follows its invocation, one of the following holds:

(1) The operation is unsuccessful according to Definition C.16. In this case, its linearization point is set to be its original linearization point, presented in Section C.1.2.
(2) The operation is successful according to Definition C.16, and some thread (not necessarily the one that executes the successful insertion) changes the state of the node allocated in line 18, in line 33. In this case, the linearization point is set to be its original linearization point as well.
(3) The operation is successful according to Definition C.16, and no thread changes the state of the node allocated in line 18, in line 33. In this case, the linearization point is set to be the insertion of a new respective volatile node to the list during recovery.

Claim C.18. *A surviving insert operation takes effect instantaneously at its linearization point.*

Proof. First, let *n* be the last volatile node allocated during a successful insert operation (according to Definition C.16). We are going to show that *n* is logically inserted into the volatile list at the operation's linearization point (presented in Definition C.17).

If some thread (not necessarily the one that executes the successful insertion) changes the state of *n* in line 33, then by Definition C.17, the operation's linearization point is this change. As proved in Section C.1.2, *n* is indeed logically inserted into the list at this point. Notice that in this case, we do not consider the insertion of a new representing node during recovery, as a logical insertion of *n* into the list. By Invariant 5 of Claim C.7, *n* is still reachable when the crash occurs. In addition, notice that as long as this node is not removed from the list, its state remains "inserted" and by Claim C.14, the state of its representing persistent node is indeed *valid* and not *removed* during recovery.

Otherwise, no thread changes *n*'s state from "intention to insert" to "inserted" before the crash event. By Definition C.16, a respective persistent node of *n* is created in line 30. From Claim C.1, *n*'s state does not change at all before the first crash and therefore, by Definition C.5, it is not logically in the list. As described in Section 4.6, during recovery, a new volatile node, representing *n*, is logically inserted into the list, and by Definition C.17, this is the linearization point of the operation in this case. Notice that by Claim C.14, the state of its representing persistent node is indeed *valid* and not *removed* during recovery, in this case as well.

When an insert operation is unsuccessful by Definition C.16, it is also unsuccessful by the original definition. From Section C.1.2, there exists a point during its execution for which a node with the given key is already logically in the list and thus, the unsuccessful operation indeed returns a correct answer. Additionally, notice that even if a representing persistent node is allocated, its state remains *valid* and *removed*, since the `create` method is only called after the volatile node is successfully inserted into the list, and the `destroy` method is only called when the state of the volatile node is either "inserted with intention to delete" or "deleted" (by Claim C.14). □

Claim C.19. *A non-surviving insert operation takes no effect.*

Proof. By Definition C.17, during a none-surviving insert operation, if a volatile node is allocated, and even if it is inserted into the list, its state remains "intention to insert" and, thus, it is not logically in the list by Definition C.5. In addition, by Definition C.17, during a non-surviving insert operation, a persistent node may be allocated, but not created (or partially created, and thus, in an *invalid* state). Therefore, during recovery, even if the persistent node is allocated, its state is either *valid* and *removed*, or *invalid*, and therefore, the represented volatile node is not inserted into the new list. □

C.2.2 Remove. We also re-define the success of a removal in the presence of crash events.

Definition C.20 (A Successful Remove Operation). Given an execution of an remove operation, we say that this operation is successful if one of the following holds:

(1) The operation returns `true`.
(2) The `result` variable is assigned with `true` in line 14, and the respective persistent node is marked as deleted in line 15 by some thread before any crash event.

The operation is unsuccessful if it returns `false`.

Definition C.21 (A Surviving Remove Operation). A remove operation is considered as a surviving operation if, before the first crash event that follows its invocation, one of the following holds:

(1) The operation is unsuccessful according to Definition C.20. In this case, its linearization point is set to be its original linearization point, presented in Section C.1.3.
(2) The operation is successful according to Definition C.20, and some thread (not necessarily the one that executes the successful removal) changes the state of the victim node in line 17. In this case, the linearization point is set to be its original linearization point as well.
(3) The operation is successful according to Definition C.20, and no thread changes the state of the victim node in line 17. In this case, the linearization point is set to be immediately after the crash event (if there is more than one such removal, they are linearized in an arbitrary order).

Claim C.22. *A surviving remove operation takes effect instantaneously at its linearization point.*

Proof. First, assume a successful remove operation (according to Definition C.20), and let *n* be the node whose state is updated from "inserted" to "inserted with intention to delete" in line 14. We are going to show that *n* is logically removed from the volatile list at the operation's linearization point (presented in Definition C.21).

If some thread (not necessarily the one that executes the successful removal) changes the state of *n* from "inserted with intention to delete" to "deleted" in line 17, then by Definition C.21, the operation's linearization point is this change. As proved in Section C.1.3, *n* is indeed logically removed from the list at this point. Notice that in this case, it is guaranteed that *n* will not be re-added into the volatile list during recovery, since by Claim C.15, it has a persistent representative, marked as deleted.

Otherwise, no thread changes *n*'s state from "inserted with intention to delete" to "deleted" before the crash event. By Definition C.20, the respective persistent node of *n* is marked as removed in line 15. From Claim C.1, *n*'s state remains "inserted with intention to delete" until the first crash event. By Claim C.9, it is logically in the list until this crash event. As described in Section 4.6, during recovery, a node representing *n* will not be inserted into the list (since its representative is marked as deleted, by Claim C.15), and in particular, will not be reachable. By Definition C.5, it will no longer be logically in the list. Therefore, it is indeed logically removed from the list at the crash event, right before its linearization point, as presented in Definition C.21.

When a remove operation is unsuccessful by Definition C.20, it is also unsuccessful by the original definition. From Section C.1.3, there exists a point during its execution for which there is no node with the given key which is logically in the list (and from Claim C.15, any persistent representative would have a *valid* and *removed* state) and thus, the unsuccessful operation indeed returns a correct answer. □

Claim C.23. *A non-surviving remove operation takes no effect.*

Proof. By Definition C.21, during a none-surviving remove operation, even if the state of the victim node becomes "inserted with intention to delete", by Definition C.21, it does not become "deleted", and no thread executes the destruction of its respective persistent node (i.e., by Claim C.14, it is still *valid* and not *removed*).

Therefore, by Definition C.5, it is still logically in the list until the crash event occurs, and during recovery, it is re-added to the new list. □

C.2.3 *Contains.*

As opposed to the insert and remove operations, a contains operation is considered as a surviving operation only when it terminates:

Definition C.24 (A Surviving Contains Operation). A contains operation is considered as a surviving operation if and only if it terminates before the first crash event occurring after its invocation. If it survives, its linearization point is set to be its original linearization point, presented in Section C.1.4.

Claim C.25. *A surviving contains operation takes effect instantaneously at its linearization point.*

Proof. Since we only consider contains operations that terminate without being interrupted by crash events, the claim follows directly from Section C.1.4 □

Claim C.26. *A non-surviving contains operation takes no effect.*

Proof. The claim follows directly from the fact that a contains operation (and in particular, an operation with no response), does not change the list. □

Theorem C.27. *The soft list is durable linearizable.*

Proof. By Definition C.17, C.21 and C.24, all the operations that are fully executed between two crashes (and some of the operations that are halted due to crash events), have a linearization point. By Claim C.18, C.22 and C.25, each operation takes effect instantaneously at its linearization point. By Claim C.19, C.23 and C.26, operations for which we did not define linearization points (non-surviving operations), do not take effect at all. In summary, the soft list is durable linearizable by definition [Izraelevitz et al. 2016]. □

C.3 Lock-Freedom

Similarly to (and following the discussion in) Section B.2, in this section we prove that in crash-free executions, at least one of the operations terminates. To derive a contradiction, assume there is some execution for which no executing operation terminates after a certain point. Notice that we can assume that no operation is invoked after this point, and that the set of running operations is finite (since there is a finite number of system threads). The rest of the proof relates to the suffix α of the execution, starting from this point.

Claim C.28. *There is a finite number of state changes during α.*

Proof. An insert operation must terminate after executing line 33 in Listing 11. Likewise, a remove operation must terminate after executing line 17 in Listing 12). In addition, any remove operation includes at most two successful state changes (in lines 14 and 17 of Listing 12). Since the number of running operations is finite by assumption, the number of state changes is finite as well. □

Claim C.29. *There is a finite number of pointer changes during α.*

Proof. The loop in lines 5–29 of Listing 11 must eventually terminate after a successful CAS execution in line 23 and therefore, there are no pointer updates in lines 20 and 23 of Listing 11. Thus, pointer updates can only occur in line 5 of Listing 9. When executing this update, the `pred` node is reachable from Claim C.2 and Invariants 1 and 3 of Claim C.7. Since `curr` is `pred`'s successor and `succ` is `curr`'s successor right before this change, `succ` is also reachable before this step. Therefore, no node becomes reachable when executing this CAS. Since this is the only possible pointer change, the list can only shrink, and the number of such pointer changes is finite. □

Theorem C.30. *The soft list is lock-free.*

Proof. From Claims C.28 and C.29, after a certain point, there are no state or pointer changes. Therefore, we consider the suffix α' of the execution that contains no state or pointer changes. Obviously, starting from this point, the list becomes stable, and does not change anymore.

Since the list is finite, from Claim C.8, every find and contains execution eventually ends. In addition, every insert and remove operation must be unsuccessful, and also terminate (since calls to the find method always terminate). We get a contradiction and therefore, the implementation is lock-free. □