הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

Efficient Lock-Free Durable Sets

Yoav Zuriel

Efficient Lock-Free Durable Sets

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Yoav Zuriel

Submitted to the Senate
of the Technion — Israel Institute of Technology
Cheshvan 5780 Haifa November 2019

This research was carried out under the supervision of Prof. Erez Petrank and Dr. Nachshon Cohen, in the Faculty of Computer Science.

The generous financial help of the Technion and the Israel Science Foundation is gratefully acknowledged.

Contents

List of Figures

Abstract

Recently Intel released a new hardware component, Non-volatile memory, promising comparable access latencies to the traditional DRAM while making the data written on it persistent, resilient to power outages. As a result, non-volatile memory is expected to co-exist or even 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. Using this approach we design two novel techniques to create durable linearizable data structures. We use these techniques to build three different kinds of sets from existing lock-free sets: linked list, hash map, and skip list. Our techniques yield the most efficient durable hash tables available today.

We ran three different tests on a 64-core AMD platform to obtain a better understanding of the different sets: scalability test, key range test, and workload test. The evaluation shows a performance improvement factor of 3.3x over the existing state-of-the-art hash map with 32 concurrent threads. To go with the new durable data structures, we extend an existing memory manager to work with persistent memory. The correctness of concurrent data structures is not trivial and thus a full correctness proof is provided for both lists proving durable linearizability and lock-freedom.

Abbreviations and Notations

BST	:	Binary Search Tree
CAS	:	Compare-and-Swap
EBR	:	Epoch-Based Reclamation
NVRAM	:	Non-Volatile RAM
RAM	:	Random Access Memory
SOFT	:	Sets with an Optimal Flushing Technique

Chapter 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 [ZS15]. 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 [IMS16]. 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.

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 [HW90]. 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 [IMS16] 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 non-volatile memory. Sets (most notably hash maps) are widely used, one example is for key-value storage [NFG+13, RKCA17, DSL10]. 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 [HS08]. 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 [CBB14].

State-of-the-art constructions of durable lock-free sets, denoted *Log-Free Data Structures*, were recently presented by [DDGZ18]. 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 [CGZ18] 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 [DDGZ18] 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 [CFL17] 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 [DDGZ18]. 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., [WH16, MCL17, CL16, LBC$^+$17, RL19, ML18, JRLR15, YMP$^+$18]. 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.

Chapter 2

Preliminaries

A *set* is an abstract data structure that maintains a collection of unique keys. It supports three basic operations: *insert*, *remove*, and *contains*. All three operation return a Boolean result to indicate if the operation succeeded. The *insert* operation adds a key to the set if the key is not already in the set and returns `true` iff the key was not previously in the set. The *remove* operation deletes the given key from the set (if the key belongs to the set), and returns `true` iff the key was in 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 consists of one word (8 bytes). Our scheme can be easily 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., [Har01, Mic02, Fra04, SS06, HS08, NM14]). 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.

Chapter 3

Overview of The Proposed Data Structures

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 [IMS16]. 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 view matches some linearization order

[1] When saying an algorithm is durable we mean the algorithm is durable linearizable [IMS16].

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 [Har01] 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.

3.1 Recovery

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

The proposed algorithms require the recovery execution to complete before further operations can be applied. Before completing recovery of the data structure on the volatile memory, the data structure is not coherent and cannot be used. This is unlike some previous algorithms, such as [FHMP18, DDGZ18], which allow the recovery and subsequent operations to run concurrently. This requirement works well in a natural setting where crashes are infrequent.

3.2 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 [Har01] 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, 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 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* technique of [DDGZ18].

3.3 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 [CGZ18].

We developed two flavors of SOFT. In the first one (presented in Section 5.1), 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 5.1.1.

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 second flavor combines the these two nodes into one, making the algorithms simpler and more space efficient, with the cost of worse performance due to increased contention. All the operations executed on the PNode below are similarly executed on the node in the combined flavor. This variant is described in Section 5.3.

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 its 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 view 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` 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 [CGZ18].

Chapter 4

The Details of the Link-Free Algorithm

In this chapter we described how to apply the *link-free* technique to a linked list (Section 4.1), a hash table (Section 4.2), and a skip list (Section 4.3).

All three sets use the same form of a node to store a single key-value pair in the set (Figure 4.1). Each node has two validity bits, two flags to reduce the number of flushes, a key, a value. The list has an additional field of *next* to point to the successor node and in skip list there is an array of next pointers for each level.

```
1 class Node{
2     atomic<byte> validityBits;
3     atomic<bool> insertFlushFlag;
4     atomic<bool> deleteFlushFlag;
5     long key;
6     long value;
7     //One or more next pointers
8 } aligned(cache-line size);
```

Figure 4.1: Link-Free Node Structure

Moreover, all three sets share the same auxiliary functions to handle the validity bits and introduced flushes. 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 to 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 flushed and avoid repeated flushing. `flipV1` and `makeValid` are two function to 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. `makeValid` can be seen in Figure 4.2

```
1  bool isValid(uchar validityBits){
2      uchar v1 = (validityBits & 0x80) >> 7;
3      uchar v2 = validityBits & 1;
4      return v1 == v2;
5  }
6
7  void makeValid(Node* n){
8      uchar oldValidity = n->validityBits.load();
9      if (isValid(oldValidity))
10         return;
11     uchar v1 = (oldValidity & 0x80) >> 7;
12     uchar newValidity = v1 + (oldValidity & 0xFE);
13     n->newValidity.store(newValidity, memory_order_release);
14 }
```

Figure 4.2: Link-Free Node's Functions

4.1 Link Free Linked List

In this section we demonstrate how to apply the principles of the link-free technique to the linked list presented by [Har01]. Building on that implementation, the list is initialized with a head with key $-\infty$, and a tail with key ∞. All the other nodes are inserted between these two, in an ascending order. Moreover, the next pointer in the node contains a marking bit to indicate a logical deletion.

4.1.1 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.

The auxiliary function `trim` (Figure 4.3) 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 (Figure 4.3) 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).

4.1.2 The contains Operation

The contains operation, based on the optimization of [HHL$^+$06], 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 (Figure 4.4), 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).

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

Figure 4.3: List Auxiliary Functions

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).

```
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 }
```

Figure 4.4: Link-Free List contains

4.1.3 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 (Chapter 6) 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 4.5). 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.

4.1.4 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 Figure 4.6). 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 (prior to line 10), valid and not removed (between lines 10 and 11), or valid and marked (after line 11). The node can never be in an inconsistent state (marked and invalid).

```
bool insert(long key, long value){
    while(true){
        Node *pred, *curr;
        pred, curr = find(key);
        if(curr->key == key){
            makeValid(curr);
            FLUSH_INSERT(curr);
            return false;
        }

        Node* newNode = allocFromArea();
        flipV1(newNode);
        atomic_thread_fence(memory_order_release);
        newNode->key = key;
        newNode->value = value;
        newNode->next.store(curr, memory_order_relaxed);
        if(pred->next.compare_exchange_strong(curr, newNode)){
            makeValid(newNode);
            FLUSH_INSERT(newNode);
            return true;
        }
    }
}
```

Figure 4.5: Link-Free List insert

```
bool remove(long key){
    bool result = false;
    while(!result){
        Node *pred, *curr;
        pred, curr = find(key);
        if(curr->key != key)
            return false;
        Node* succ = getRef(curr->next.load());
        Node* markedSucc = mark(succ);
        makeValid(curr);
        result = curr->next.compare_exchange_strong(succ, markedSucc);
    }
    trim(pred, curr);
    return true;
}
```

Figure 4.6: Link-Free List remove

4.2 Link Free Hash Table

The hash table implementation relies heavily on the list (the same applies to SOFT hash table). It has a fixed sized array where each entry is simply a link-free list. Each hash table operation is delegated to the corresponding link-free list, using a standard hash function. The code of all three operations can be seen in Figure 4.7.

```
1  bool insert(long key, long value){
2      LinkFreeList bucket = table[hash(key)];
3      return bucket.insert(key, value);
4  }
5
6  bool remove(long key){
7      LinkFreeList bucket = table[hash(key)];
8      return bucket.remove(key);
9  }
10
11 bool contains(long key){
12     LinkFreeList bucket = table[hash(key)];
13     return bucket.contains(key);
14 }
```

Figure 4.7: Link-Free Hash Table Operations

4.3 Link-Free Skip List

We extend the lock-free skip list presented by Fraser [Fra04] to use the link-free techniques. To the link-free node we added an integer to store the number of levels the node is in. Also, we included more next pointers for each of the levels, and as per the original algorithm, all of the pointers in all of the levels can be marked.

Since only the next pointer in the lowest level indicates the belonging of a key to the set, only after marking this pointer, the node is flushed. Note that, we need to flush a single cache line regardless of the skip list node actual size, since the missing links will be reconstructed during a recovery.

The original skip list uses a few auxiliary that we use without providing an implementation. First, in the skip list we use the same bit operations as in the list, `mark`, `getRef` and `isMarked`, and the do the same in the context of the skip list as well.

Second, the skip list has its own version of `find`, where the skip list is traversed and a node can be found in logarithmic time and a list of the node's predecessors and successors is filled along the function's execution (one pair per level). `find` returns whether the key of the successor on the lowest level equals to the requested key. There are three variant of the find function, `cleanupFind` which flushes and tries to physically remove nodes which are marked at level 0. The second and third variant are `noCleanupFind` and `noCleanupFindSuccs` which only traverse the list without physically unlinking, and as a result of that, without flushing any nodes. The difference between the two is that `noCleanupFindSuccs` does not return a list of predecessors, only a list of successors.

Another useful function is `markNode` which is called to mark all of the node's pointers. The marking is done in a descending order, and it returns whether the current thread marked the pointer on the lowest level (Section 4.3.3).

4.3.1 The contains Operation

The algorithm (Figure 4.8) is similar to the one presented in [Fra04] and it is wait-free as well. We added the condition in lines 8 - 11 to check if the node we are looking for is deleted, and if we are looking at the lowest level. If so, the node is seen as out of the set, so we flush its removal and return `false`. The second addition was lines 16, 17 where a key is found and since it is out of the loop is cannot be marked. Before returning `true`, we make it valid and flushed, so the existence persists.

4.3.2 The insert Operation

The insert operation (Figure 4.9) starts by looking for the given key (line 5). If the key is found, before returning `false`, the key's node is made valid and persisted. Otherwise, a new node is allocated, it is made invalid by flipping its first validity bit, the node's fields are initialized and all of the next pointers are written using the successors list.

```
1  bool contains(long key){
2      Node *pred = this->head;
3      for (int i = MAX_LEVEL - 1; i >= 0; i--){
4          Node *curr = getRef(pred->next[i].load());
5          while (curr->key < key || isMarked(curr->next[i].load())){
6              if (!isMarked(curr->next[i].load()))
7                  pred = curr;
8              else if (i == 0 && curr->key == key){
9                  FLUSH_DELETE(curr);
10                 return false;
11             }
12             curr = getRef(curr->next[i].load());
13         }
14
15         if (curr->key == key){
16             makeValid(curr);
17             FLUSH_INSERT(curr);
18             return true;
19         }
20     }
21     return false;
22 }
```

Figure 4.8: Link-Free Skip List contains

In line 22 the thread tries to link the new node to the skip list at level 0. If it fails, the whole operation is restarted. A successful CAS makes the node in the set, so other concurrent operations now acknowledge it.

The node is made valid and persisted and finally the operations ends by CASing all of the other next pointers to the new node. Note that CASing the other pointers is not mandatory since the node is connected on level 0, hence it is in the set. The other pointers are used for improved performance. Because of that reason, if one of the new node's pointer is marked (line 34), i.e., it is in the middle of deletion, the thread may return `true` safely.

4.3.3 The remove Operation

In order to remove a node (Figure 4.10), `noCleanupFindSuccs` is invoked (line 3). If the key is not found, the thread return `false` and exits. Otherwise, it tries to mark the node's next pointers by calling `markNode` (line 8). The function returns `true` to only one thread which logically removed the node, and `false` to all the others. All the concurrent removes flush the removed node before returning, but only the thread which actually removed the node physically unlinks in by calling `cleanupFind` (line 11).

```
bool insert(long key, long value){
    Node *preds[MAX_LEVEL], *succs[MAX_LEVEL];

    while(true){
        if (noCleanupFind(key, preds, succs)){
            makeValid(succs[0]);
            FLUSH_INSERT(succs[0]);
            return false;
        }

        Node *newNode = allocNodeFromArea();
        flipV1(newNode);
        std::atomic_thread_fence(std::memory_order_release);
        newNode->key = key;
        newNode->value = value;
        newNode->topLevel = getRandomLevel();

        for (int i = 0; i < newNode->topLevel; i++)
            newNode->next[i].store(succs[i], memory_order_relaxed);

        if (preds[0]->next[0].compare_exhange_strong(getRef(succs[0]),
        newNode))
            break;
    }

    makeValid(newNode);
    FLUSH_INSERT(newNode);

    for (int i = 1; i < newNode->topLevel; i++){
        while (true){
            Node *pred = preds[i];
            Node *succ = succs[i];
            next = newNode->next[i].load();
            if (isMarked(next))
                return true;
            if (pred->next[i].compare_exchange_strong(succ, newNode))
                break;
            cleanupFind(key, preds, succs);
        }
    }
    return true;
}
```

Figure 4.9: Link-Free Skip List insert

```
1  bool remove(long k){
2      Node *succs[MAX_LEVEL];
3      if (!noCleanupFindSuccs(key, succs))
4          return false;
5
6      Node *node = succs[0];
7      makeValid(node);
8      bool result = markNode(node);
9      FLUSH_DELETE(node);
10     if (result)
11         cleanupFind(key, NULL, NULL);
12     return result;
13 }
```

Figure 4.10: Link-Free Skip List remove

4.4 Recovery

The validity scheme we use helps us determine whether a node was linked to the data structure 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 data structure 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 set. 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 data in the nodes is already stored in the NVRAM. Note that this scanning can be done in a parallel manner, if each thread scans its own areas and the insertion is done in a parallel manner.

Chapter 5

The Details of SOFT

As discussed in Section 3.3, SOFT achieves the lower bound on the number of fence instructions used. We adapted this technique to a linked list (Section 5.1), a hash table (Section 5.2), and a skip list (Section 5.3).

The transformation from a simple lock-free data structure to a SOFT one, is not as easy as it was with the link-free algorithm. Since the number of `psync` operations should be minimal, a different approach is taken, where the linearization point occurs only after persisting data on the NVRAM. Because of that we can guarantee that some linearization survives a crash without too many `psync` operations without considering many dependency scenarios. This guarantee comes with the cost of using more CAS primitives along the different operations, more while loops and algorithmic complication.

5.1 SOFT Linked List

The second algorithm we present is SOFT, which achieves the lower bound on the number of `psync` operations. It does so by dividing each update operation into two stages: intention and completion. By doing so, a thread triggers helping mechanisms by other threads, while not changing the logical state of the data structure. In this section, we start by describing the nodes of the SOFT list (Sections 5.1.1 and 5.1.2), then we discuss the implementation details of each set operation (Sections 5.1.3, 5.1.4 and 5.1.5).

5.1.1 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 is provided in Figure 5.1.

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

Figure 5.1: PNode

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 Figure 5.2.

```
1  bool PNode::alloc(){
2      return !validStart.load();
3  }
4
5  void PNode::create(long key, long value, bool pValidity){
6      validStart.store(pValidity, memory_order_relaxed);
7      atomic_thread_fence(memory_order_release);
8      this->key.store(key, memory_order_relaxed);
9      this->value.store(value, memory_order_relaxed);
10     validEnd.store(pValidity, memory_order_release);
11     psync(this);
12 }
13
14 void PNode::destroy(bool pValidity){
15     deleted.store(pValidity, memory_order_release);
16     psync(this);
17 }
```

Figure 5.2: PNode Member Functions

5.1.2 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 5.1.1) and *pValidity*, a Boolean flag indicating the pValidity of the persistent node. The structure of the volatile node appears in Figure 5.3.

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

Figure 5.3: Volatile Node

Similar to the lock-free linked list algorithm by [Har01], 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 3.3). We assume standard methods for handling pointers with embedded state (lines 2 – 7 Figure 5.5). In addition, we use `trim` and `find` to physically unlink removed nodes and find the relevant window, respectively (Figure 5.4). 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.

```
1  bool trim(Node *pred, Node *curr) {
2     state predState = getState(curr);
3     Node *currRef = getRef(curr), *succ = getRef(currRef->next.load());
4     succ = createRef(succ, predState);
5     return pred->next.compare_exchange_strong(curr, succ);
6  }
7
8  Node*, Node* find(long key, state *currStatePtr){
9     Node *pred = head, *curr = pred->next.load();
10    Node *currRef = getRef(curr);
11    state predState = getState(curr), cState;
12    while (true){
13       Node *succ = currRef->next.load();
14       Node *succRef = getRef(succ);
15       cState = getState(succ);
16       if (cState != DELETED){
17          if (currRef->key >= key)
18             break;
19          pred = currRef;
20          predState = cState;
21       }
22       else
23          trim(pred, curr);
24       curr = createRef(succRef, predState);
25       currRef = succRef;
26    }
27    *currStatePtr = cState;
28    return pred, curr;
29 }
```

Figure 5.4: find and trim

5.1.3 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`.

5.1.4 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 Figure 5.6 and is discussed below.

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

Figure 5.5: SOFT List contains

Similar to link-free, persistent nodes are allocated from a durable area using the method `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 a volatile node is marked as removed, it is always safe to unlink it from the data structure 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 24) 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, where the node is fully inserted.

The helping part starts by initializing the PNode of the appropriate node (line 31). Afterwards, all the threads try to complete the insertion and make it visible by changing the state of the new node to "inserted" (line 34). Finally, the thread returns `true` or `false` depending on the path taken.

```
bool insert(long key, long value){
    Node *pred, *curr, *currRef, *resultNode;
    state predState, currState;

    while(true){
        pred, curr = find(key, &currState);
        currRef = getRef(curr);
        predState = getState(curr);
        bool result = false;
        if(currRef->key == key){
            if(currState != INTEND_TO_INSERT)
                return false;
            resultNode = currRef;
            break;
        }
        else{
            PNode* newPNode = allocFromArea();
            bool pValidity = newPNode->alloc();
            Node* newNode = new Node(key, value, newPNode, pValidity);
            newNode->next.store(createRef(currRef, INTEND_TO_INSERT),
                memory_order_relaxed);

            if(!pred->next.compare_exchange_strong(curr,
                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;
}
```

Figure 5.6: SOFT List insert

5.1.5 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.

```
bool remove(long key){
    bool result = false;
    Node *pred, *curr;
    state predState, currState;
    pred, curr = find(key, &currState);
    Node* currRef = getRef(curr);
    predState = getState(curr);
    if(currRef->key != key)
        return false;
    if(currState == INTEND_TO_INSERT)
        return false;

    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;
}
```

Figure 5.7: SOFT List remove

5.2 SOFT Hash Table

The SOFT hash table implementation relies heavily on the list as well. It has a fixed sized array where each entry is simply a SOFT list. Each hash table operation is delegated to the corresponding SOFT list, using a standard hash function. The code of all three operations can be seen in Figure 5.8.

```
bool insert(long key, long value){
   SOFTList bucket = table[hash(key)];
   return bucket.insert(key, value);
}

bool remove(long key){
   SOFTList bucket = table[hash(key)];
   return bucket.remove(key);
}

bool contains(long key){
   SOFTList bucket = table[hash(key)];
   return bucket.contains(key);
}
```

Figure 5.8: SOFT Hash Table Operations

5.3 SOFT Skip List

Similar to link-free, we can use SOFT and create a durable lock-free skip list, with minimal use of fences per operations (following the lower bound [CGZ18]). The node used is different from the one presented in the list section. The new description of this node appears in Section 5.3.1.

We continue using the states on the next pointers in the skip list as well. However, since the pointers at higher levels are used for performance only, they do not have to go through the different states. Only the pointer at the lowest level has to transition from "intention to insert" through "inserted" and "inserted with intention to delete" to become eventually "deleted". Similar to before, a single cache-line consists all the vital data (key, value, etc.) and the pointer of level 0, thus we use a single `psync` operation to "flush" a node. The other pointers are reconstructed during recovery, on demand.

5.3.1 SOFT Skip List Node

In this section we describe a new variant to the node used in SOFT. The main difference is the fact this node encapsulate both the volatile and persistent node presented in Section 5.1. This node like the list's volatile node has a key, a value, a validity flag and several *next* pointers, and like the persistent node, it has a three-flag validity scheme.

```
1 class Node{
2    bool pValidity;
3    atomic<bool> validStart, validEnd, deleted;
4
5    long key;
6    long value;
7    int topLevel;
8    atomic<Node*>[] nexts;
9 } aligned(cache-line size);
```

Figure 5.9: SOFT Skip List Node Structure

The insertion of a new key resembles the insertion of link-free. The node is made invalid (by flipping `validStart`), and only then it is initialized with the "intention to insert" state on the pointer in level 0. Afterwards, the node is linked to the lower level (in an invalid state), becomes valid (by equating `validEnd` to `validStart`, and persisted. Unlike link-free, the node is still not considered as a part of the list as the state on its pointer is "intention to insert". To complete the insertion, the state becomes "inserted" (using a CAS).

The auxiliary functions of the list's persistent node also change slightly (presented in Figure 5.10). In addition, we added a new function to initialize a skip list node given a key and value, `createNode`. This function works by making the new node invalid and initializing its values. The node's `topLevel` field is random, and the validity is acquired using the `alloc` auxiliary function.

```
1  //Pseudo-code for managing state pointers.
2  #def getRef(sPointer) {sPointer.ptr}
3  #def createRef(address, state) {.ptr=address, .state=state}
4  #def getState(sPointer) {sPointer.state}
5  #def stateCAS(*sPointer, oldState, newState)
6  {CAS(sPointer.state, oldState, newState);}
7
8  bool alloc() {
9     return !validStart.load();
10 }
11
12 void help() {
13    validEnd.store(pValidity, memory_order_release);
14    psync(this);
15 }
16
17 void destroy() {
18    deleted.store(pValidity, memory_order_release);
19    psync(this);
20 }
21
22 void createNode(long key, long value){
23     bool validity = this->alloc();
24    this->validStart.store(validity, memory_order_relaxed);
25    atomic_thread_fence(memory_order_release);
26    this->key = key;
27    this->value = value;
28    this->topLevel = get_random_level();
29    this->pValidity = validity;
30 }
```

Figure 5.10: SOFT Node Auxiliary Functions

The SOFT skip list uses auxiliary functions such as `noCleanupFind` and `cleanupFind`, similar to the ones of the link-free skip list (Section 4.3). Just like in the SOFT list `find` function, the skip list ones also return the states of the different pointers, the states of the previous pointers are stores with the successor pointers, and the states of the successor pointers are stored in an array.

5.3.2 The contains Operation

Since SOFT matches the lower bound, there is no need to execute a single `psync` during the contains operation, making the code almost identical to the original. The only difference is the condition to check and make sure a node is indeed in the skip list. This condition is encapsulated in `isOut`, which checks the state of the pointer in the current level, if it is "intention to insert" of "deleted" is it considered out of the skip list.

```
1  bool contains(long key){
2      Node *pred = this->head, *curr;
3     for (int i = MAX_LEVEL - 1; i >= 0; i--){
4        curr = getRef(pred->next[i].load());
5        while (curr->key < key || isOut(curr->next[i].load())){
6           if (!isOut(curr->next[i].load()))
7              pred = curr;
8           curr = getRef(curr->next[i].load());
9        }
10       if (curr->key == key)
11          return true;
12    }
13    return false;
14 }
```

Figure 5.11: SOFT Skip List contains

5.3.3 The insert Operation

To insert a new node to the SOFT skip list, the thread first searches the skip list. If a node with the same key is found, denoted n with "intention to insert" state (we remind the reader that a node returned from a find method cannot be in a "deleted" state), the operation can fail and return `false`. In case the state of n is "intention to insert", the current thread cannot insert its node, however, n is not in the list yet. Thus, the current thread help to complete n's insertion. It makes the node valid and executes a `psync`, by calling `help` (line 12 Figure 5.12) and then it changes the state of n to be "inserted".

Otherwise, there is no node with the given key in the data structure. The thread then allocates a new node, makes it invalid and initializes the node's fields, including the *next* pointers. Note that the pointer on the lowest level is infused with the "intention

to insert" state. Afterwards, the thread tries to link its node to the skip list, if it fails, the whole operation is restarted. Following a successful linking, the node is made valid and it is written to the NVRAM using `help` (line 33), and the state of the node changes to "inserted" to complete its insertion. Finally, the rest of the pointers in the other levels are set. The values of the *next* pointers on the higher levels (all levels except the lowest one), are only used for performance and not for correctness, so they can be set in a non-atomic way.

5.3.4 The remove Operation

To remove a node, the thread finds it first. If there is no node with the given key, or its state is "intention to insert" then the operation can fail immediately. If such a node n exists, the thread changes n's state to "intention to delete", calls `destroy` to make n deleted (all three validity flags match), and finally n's state changes to "deleted".

5.4 Recovery

In SOFT list only 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.

Similarly, in the SOFT skip list, the next pointers are not written back the the NVRAM, so the intentions may not be available during recovery. Just like SOFT list, however, the procedure distinguishes valid nodes from invalid ones using the validity scheme.

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".

In order to recover a SOFT skip list, a similar technique is used. We create an empty skip list to which all the surviving nodes will be insert to. The durable areas are scanned, and valid and not deleted nodes are linked back to the skip list. Note that, we can choose a new level for surviving nodes if we wish (to better balance the new skip list). It is not necessary, however, since this field must be coherent for the node to be valid.

Similar to link-free, no `psync` operations are used to link the nodes since the data already persisted in the NVRAM. Invalid or deleted nodes are sent to the memory manager for reclamation.

```
bool insert(long key, long value){
   Node *newNode, *pred, *succ, *next;
   Node *preds[MAX_LEVEL], *succs[MAX_LEVEL];
   state succStates[MAX_LEVEL];
   bool result;

   while(true){
      if (noCleanupFind(key, preds, succs, succStates)){
         if (succStates[0] != INTEND_TO_INSERT)
            return false;
         newNode = getRef(succs[0]);
         newNode->help();
         while(getState(newNode->next[0].load()) == INTEND_TO_INSERT)
             stateCAS(newNode->next, INTEND_TO_INSERT, INSERTED);
         return false;
      }
      newNode = allocFromArea();
      newNode->createNode(key, value);

      Node *succRef = getRef(succs[0]);
      newNode->next[0].store(createRef(succRef, INTEND_TO_INSERT),
                     memory_order_release);
      for (int i = 1; i < newNode->topLevel; i++){
         Node *currRef = getRef(succs[i]);
         newNode->next[i].store(createRef(currRef, INSERTED),
         memory_order_relax);
      }

      Node *after = createRef(newNode, getState(succs[0]));
      if (preds[0]->next[0].compare_exchange_strong(succs[0], after))
         break;
    }
   newNode->help();
   while(getState(newNode->next[0].load()) == INTEND_TO_INSERT)
       stateCAS(newNode->next, INTEND_TO_INSERT, INSERTED);

   for (int i = 1; i < newNode->topLevel; i++){
      while (true){
         pred = preds[i];
         succ = succs[i];
         next = newNode->next[i].load();
         if (isOut(next))
            return true;
         if (pred->next[i].compare_exchange_strong(succ, newNode))
            break;
         cleanupFind(key, preds, succs, succStates);
      }
   }
   return true;
}
```

Figure 5.12: SOFT Skip List insert

```
1  bool remove(long k){
2     Node *succs[MAX_LEVEL];
3     state succStates[MAX_LEVEL];
4     if (!noCleanupFindSuccs(key, succs, succStates))
5        return false;
6
7     Node *node = getRef(succs[0]);
8     if (isOut(succStates[0]))
9        return false;
10    bool result = markNodes(node);
11    node->destroy();
12    while(getState(currRef->next[0].load()) == INTEND_TO_DELETE)
13       stateCAS(node->next, INTEND_TO_DELETE, DELETED);
14
15    if (result){
16       cleanupFind(key, nullptr, nullptr, nullptr);
17       return true;
18    }
19    return false;
20 }
```

Figure 5.13: SOFT Skip List remove

Chapter 6

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 [Mic02] and to use-after-free.

To maintain the lock-freedom of our algorithms, lock-free memory reclamation schemes can be used (e.g., [Coh18, Mic04, ALMS17, BGHZ16, DHK16, CP15, Bro15]). 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) [Fra04] 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 [DGT15].

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.

Chapter 7

Measurements

We ran the measurements of the link-free and SOFT algorithms and compared them to the state-of-the-art set algorithm proposed by [DDGZ18]. 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 [VTS11, WJ14, CBB14, APD15, SDUP15, KPS+16, CFL17, DDGZ18, FHMP18, CAAL19, BDBFW19], we measured the performance using a DRAM. NVRAM is expected to be somewhat slower than DRAM [APD15, VTS11, WJ14]. Nevertheless, we assume that data is durable once it reaches the memory controller[1]. 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 [Int19], so an additional store fence is not required (unlike the `clflushopt` instruction, which does require a fence). [DDGZ18] 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).

7.1 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.

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

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 [HS08]), 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 one.

The results for the throughput test are displayed in Figure 7.1. 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 relative improvement over the log-free set is shown (the y axis is the factor of improvement).

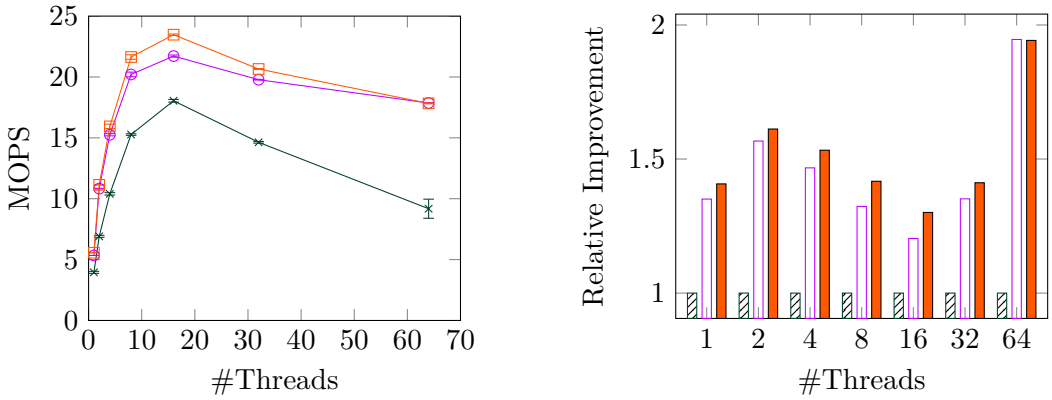
In Figures 7.1a and 7.1b, we can see the results for the shorter and longer lists. When the key range is 256 keys, all algorithms experience a peak with 16 threads and a slow decrease towards 64 threads. For a single thread, SOFT and link-free outperform log-free by 40% and 35%, respectively, for 16 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.

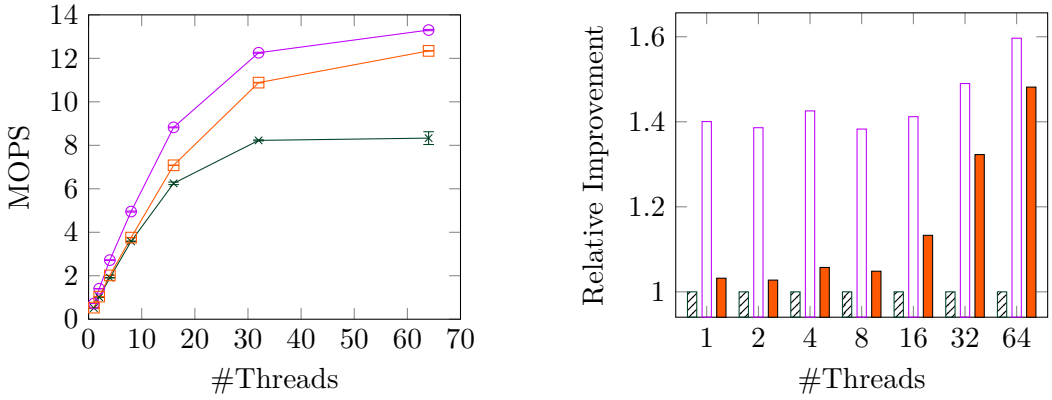
For a longer list (Figure 7.1b), all the compared lists scale with the additional threads. When the number of available keys is bigger, most of the time is spent on traversing the list; hence, more threads imply more concurrent traversals and more operations.

As can be seen in the graph, link-free outperforms both SOFT and log-free by a considerable difference. In contrast to Figure 7.1a, 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.

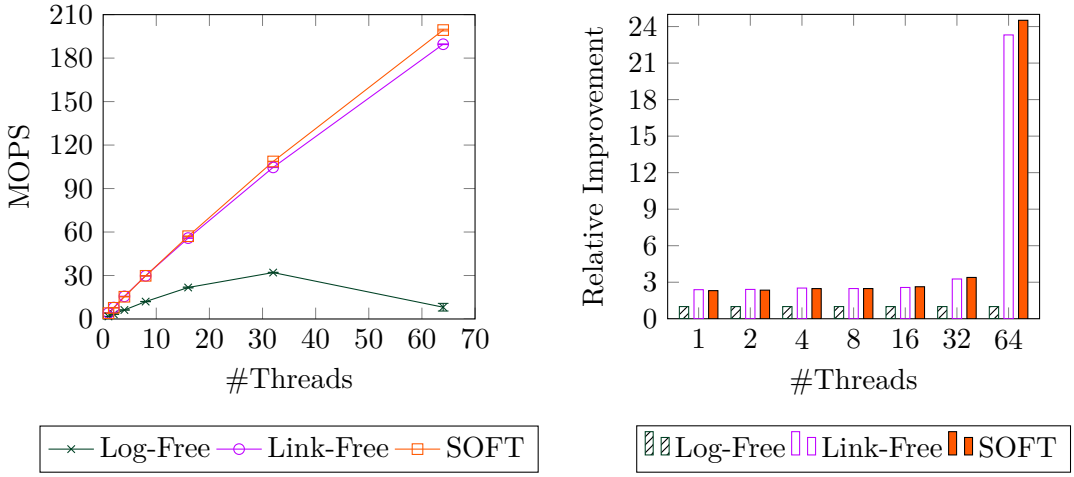
Moreover, with higher contention, a node might be flushed more than once in link-free. As mentioned, link-free prevents redundant `psync` operations using a flag after the first necessary `psync`. In a case where multiple threads operate on the same key, it might be flushed more than needed. So, when 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



(a) List's Throughput with Range of 256

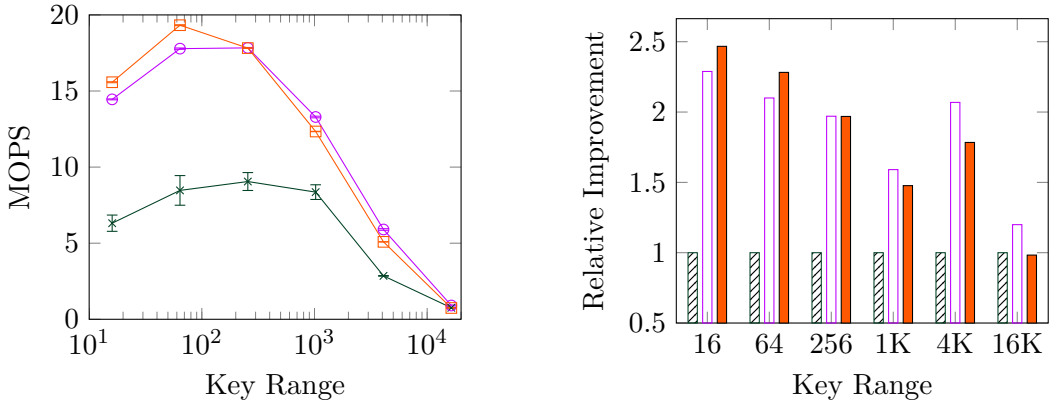


(b) List's Throughput with Range of 1024

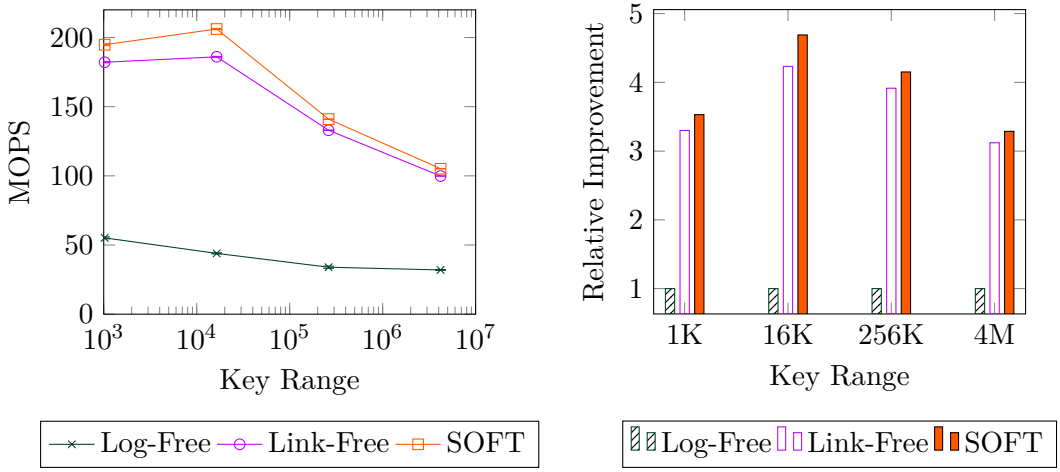


(c) Hash Table's Throughput with Range of 1M

Figure 7.1: Throughput as a Function of the #Threads



(a) List's Throughput



(b) Hash Tables' Throughput

Figure 7.2: Throughput as a Function of Key Range

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 7.1c. 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 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 7.2a 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.

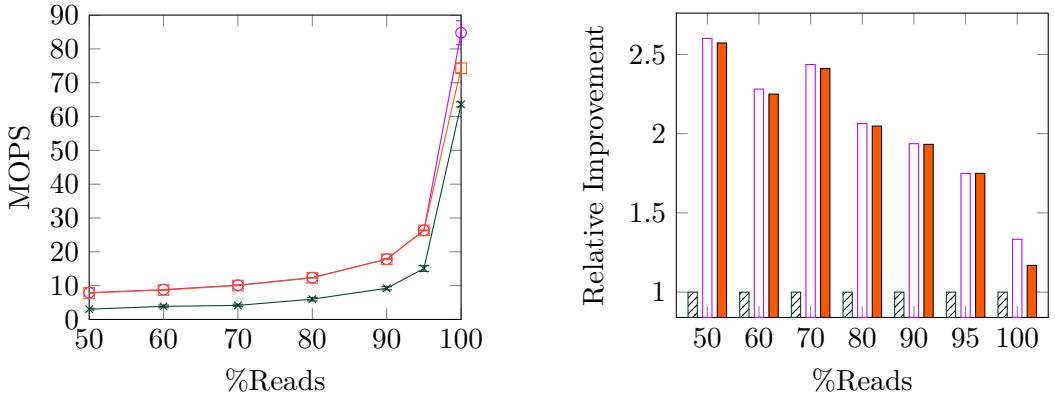
Figure 7.2b depicts the performance and the relative improvement of the three hash tables. As explained above, this test was run with 32 threads. As predicted, the performance of all hash tables worsens as the range grows. This may be attributed to reduced locality. For 1K distinct keys, SOFT outperforms 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) [CST+10] 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.

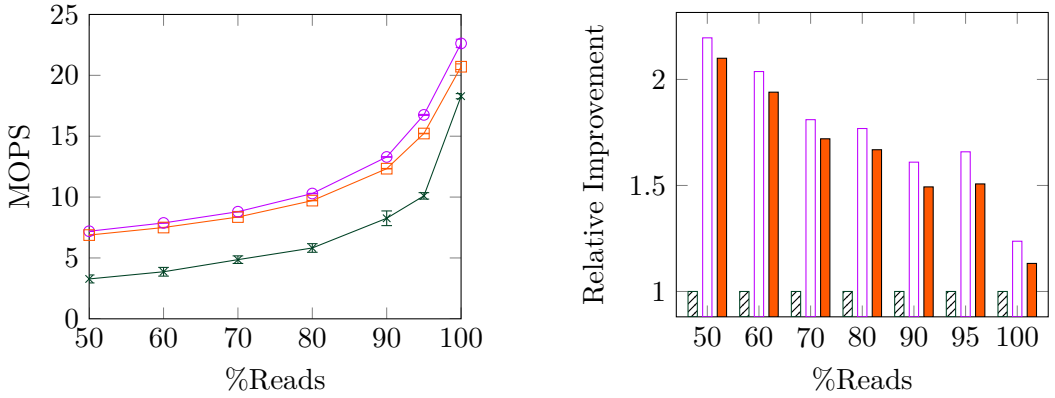
The lists (Figures 7.3a and 7.3b) all behaved similarly to one another. For both ranges, link-free performed slightly better than SOFT. Link-free is superior to SOFT since the high amount of threads increases the contention, which increases the cost of the additional CAS-es used in SOFT. Also, a higher 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.

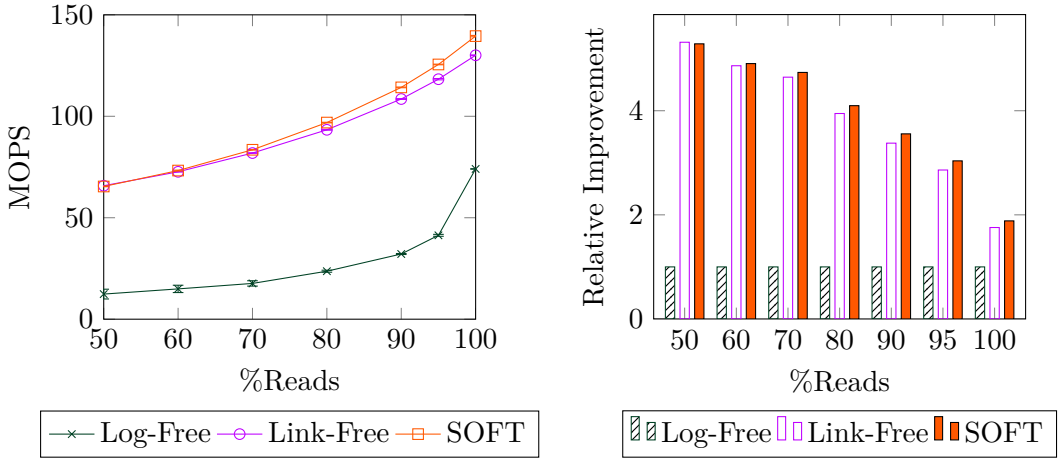
The trend of both graphs can be justified by a few reasons. First, all algorithms use the least amount of `psync` operations in the read operations. SOFT does not use any, link-free uses at most one, and log-free uses at most two. Moreover, reads are



(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

Figure 7.3: Throughput as a Function of the Percentage of Reads

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 7.3c, 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.

Chapter 8

Related Work

There has been a lot of research focused on adapting specific concurrent data structures to durable ones [SDUP15, NIK+17, FHMP18, DDGZ18]. Some researchers developed techniques to modify general objects so that they are durable linearizable [CCA+12, VTS11, IMS16, KPS+16, AB16, CGZ18]

[VTS11, CCA+12, KPS+16] 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.

[IMS16] 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

[CFL17] presented a sequential durable hash table that uses only one `psync` per update and none for reads, achieving the lower bound proven by [CGZ18]. 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.

[NIK+17] 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.

[FHMP18] presented three variations of a durable lock-free queue. The first guarantees durable linearizability [IMS16], the second queue guarantees *detectable execution* [FHMP18], which is a stronger guarantee than durable linearizability, and the third

queue guarantees buffered durable linearizability [IMS16]. The queue is inherently different from a set since it maintains an order between individual keys.

[CGZ18] 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.

[DDGZ18] 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 [DGT13]. 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.

Chapter 9

Conclusion

In this work we presented two techniques for durable lock-free sets, link-free and SOFT, and implemented three different sets, linked list, hash map and skip list. These two techniques 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 `psync` operations to the minimum theoretical value, at the expense of algorithmic complication and higher (volatile) synchronization, and thus introduces a new trade-off to consider when designing durable data structures. 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.

We believe these techniques can be further used to create more durable data structures from non-durable ones. Note that our techniques cannot be applied in an automatic manner, but rather with careful inspection to remove redundant `psync` operations.

Appendix A

Link Free Correctness

We start by proving some basic list invariants. In Section A.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 A.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 A.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 A.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 4.5, satisfying $newNode = n$.

Definition A.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 A.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 4.4, line 6, 12 or 18 of Listing 4.5, or in line 10 or 11 of listing 4.6. As explained is Section 4.1.1, 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 4.6), 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 4.5. ■

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

Proof. Let n be a marked node. From Claim A.4, it cannot be unmarked. Besides when marked in line 11 of listing 4.6, n's *next* pointer can only change during a successful CAS execution in line 4 of Listing 4.3 or line 17 of Listing 4.5. 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 A.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 A.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 A.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 4.5, 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 A.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 A.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 4.3, or in line 16 or 17 of Listing 4.5. If s is the execution of line 4 in Listing 4.3 or line 16 in Listing 4.5, 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 4.5, then n_2 is not an infant by Definition A.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 A.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 A.4, its state must be valid. n_2's state can become valid only in line 11 of Listing 4.4, in line 6 or 18 of Listing 4.5, or in line 10 of Listing 4.6. 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 4.5, 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 A.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 A.1 is the longest. The node n_1 can only become unreachable if the current step is the execution of line 4 in Listing 4.3, and if n_1 is marked and then flushed in line 2 of Listing 4.3. 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 4.3, and line 16 and 17 of Listing 4.5) preserve keys order (notice the halting condition in line 11 of Listing 4.3), 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 4.3, 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 4.5. 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 4.3 or in line 17 of Listing 4.5. 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 A.1, `tail` is reachable from n_1 in this case as well. ■

Claim A.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 A.7, the volatile list is always sorted by the nodes' keys and no key ever appears twice. By Claim A.6 and Invariant 3 of Claim A.7, the `head` and `tail` sentinel nodes are always members of the list, and by Invariant 5 of Claim A.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 A.9 below.

Definition A.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 A.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 A.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 A.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 A.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 A.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 A.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 4.3, line 8 of Listing 4.4, or implicitly flushed yet, and in particular, it has not become unreachable in line 4 of Listing 4.3 yet (according to the proof of Claim A.7, it cannot become unreachable in other scenarios). Therefore, n is still reachable in this case as well. ■

Notice that Claim A.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 A.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 A.8 and A.11. ■

Claim A.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 A.7, it is not an infant, and from Invariant 3 of Claim A.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 A.5, n_1 is unmarked at this point and thus, from Invariant 3 of Claim A.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 A.14. *Let there be an insert execution that returns false in line 8 (Listing 4.5), 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 A.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 A.4), no later than the execution of line 7, it is either persistently in the list (by Definition A.9), or becomes marked before its flush. In both cases, the claim holds. ■

Claim A.15. *Let n_2 be a node which is assigned into the `curr` variable in line 4 of Listing 4.4, 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 A.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 A.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. ■

A.1 Durable Linearizability

We use the notion of *durable linearizability* [IMS16] for correctness. The recovery procedure, executed after a crash (and described in Section 4.4), 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 A.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 A.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 A.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 A.17.

Claim A.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 A.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 A.16, it is valid, and either not marked, or marked and not flushed yet. By Invariant 3 of Claim A.7, n_1 is reachable. By Claim A.8, there cannot be two reachable nodes with the same key, and therefore, n_2 is not reachable, and by Invariant 3 of Claim A.7, it is either not valid, or marked and flushed. In both cases, it is not a set member according to Definition A.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.

A.1.1 Insert

Before defining the conditions for the survival of an insert operation, we need to redefine the success of an insertion in the presence of crash events.

Definition A.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 A.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 A.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 A.16 (chosen arbitrarily).

2. The operation is successful according to Definition A.18, and the node allocated in line 11 becomes persistently in the list (see Definition A.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 A.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 A.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 A.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 A.19, and therefore, the (unsuccessful) insert operation indeed takes effect at this point. According to Claim A.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 A.16, m is indeed a set member, and we are done. In the second scenario, it is guaranteed by Claim A.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 A.16, and the claim holds.

2. Suppose that the operation is successful according to Definition A.18, and the node allocated in line 11 becomes persistently in the list (see Definition A.9) before the crash event. By Definition A.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 A.18, and the node allocated in line 11 does not become persistently in the list before the first crash event. By Definition A.18, it becomes valid, then marked, and then flushed, during the execution, and before any crash event. By Definition A.16, it becomes a set member when its state becomes valid and thus, the operation indeed takes effect at its defined linearization point. ■

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

Proof. By Definition A.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 A.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. ■

A.1.2 Remove

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

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

Claim A.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 A.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 A.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 A.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 A.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 A.23) indeed removes this node from the set, according to Definition A.16. Otherwise, the mentioned flush is the first flush of the victim node, and according to Definition A.16, it removes it from the set in this case as well. ■

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

Proof. By Definition A.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 A.16, whether it is originally a set member or not, it is not removed from the set. ■

A.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 A.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 A.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 A.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 A.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 A.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 A.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 A.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 A.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 A.8, there does not exist a reachable node with the given key at this point. By Claim A.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 A.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 A.15), at the latest, it is flushed as a marked node in line 2 of Listing 4.3. Therefore, the guaranteed point exists in this case as well. By Claim A.8 and A.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 A.1. *The link-free list is durable linearizable.*

Proof. By Definition A.19, A.23 and A.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 A.20, A.24 and A.27, each operation takes effect instantaneously at its linearization point. By Claim A.21 and A.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 [IMS16]. ■

A.2 Lock-Freedom

A.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 A.28. *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 4.6. 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 4.5, we can assume that the number of state changes in line 10 of Listing 4.6 is finite as well. ■

Claim A.29. *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 4.3 or line 17 of Listing 4.5. A state change in line 17 of Listing 4.5 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 4.3. Since there is a finite number of state changes of reachable nodes during α (by Claim A.28), the number of marked nodes is bounded and thus, there is a finite number of pointer changes of reachable nodes during α. ■

Theorem A.2. *The link-free list is lock-free.*

Proof. From Claims A.28 and A.29, 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 A.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. ■

Appendix B

SOFT Correctness

In this chapter 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 B.1 we prove the linearizability of our implementation when there are no crash events, followed by a durable linearizability proof in Section B.2. Finally, we show our implementation is lock-free in Section B.3.

Claim B.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 34 of Listing 5.6, or in line 14 or 17 of Listing 5.7. 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 5.4 and line 24 of Listing 5.6), the state stays unchanged. ■

Claim B.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 5.4 or in line 24 of Listing 5.6. 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 B.1, its state cannot become "deleted" and change again afterwards). Notice that we deal with state changes in Claim B.1. In this claim we refer only to reference changes. ■

Claim B.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 B.1, a node's state can change either in line 34 of Listing 5.6, or in line 14 or 17 of Listing 5.7. 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 B.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 $0 \leq i < k$, 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 B.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 B.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 24 in Listing 5.6, satisfying $newNode = n$.

Claim B.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 24 in Listing 5.6, 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 5.4, in line 21 of Listing 5.6, or in line 24 of Listing 5.6. 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 B.6.

2. Assume that the next step will execute line 24 of Listing 5.6, having $newNode = n_2$. Assume by contradiction that n_2 is not an infant. Since the CAS in line 24 can only be executed on nodes created in line 19, 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 34 of Listing 5.6, 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_1 was an infant before the current step, then the current step is the execution of line 24 in Listing 5.6, making n_1 the successor of some node which is reachable by assumption. n_1 is reachable in this case. Otherwise, by assumption and Claim B.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 B.4 is the longest. The node n_1 can only become unreachable if the current step is the execution of line 5 in Listing 5.4, 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 5.4 and lines 21 and 24 in Listing 5.6) preserve keys order (notice the halting condition in line 17 of Listing 5.4), 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 5.4, 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 24 of Listing 5.6) 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 5.4 or in line 24 of Listing 5.6. 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.4, `tail` is reachable from n_1 in this case as well. ■

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

Claim B.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 B.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 B.5. When n's state was changed from "intend to insert" to "inserted" in line 34 of Listing 5.6, it must have had a predecessor. From Invariant 1 of Claim B.7, it is not an infant. From Invariant 3 of Claim B.7, it is reachable. ■

B.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.

B.1.1 Find

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

Claim B.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 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'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 B.2, n_1's state is not "deleted" at this point and thus, from Invariant 3 of Claim B.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. ■

B.1.2 Insert

Let n be the volatile node created during a successful execution of the insert operation (line 19 in Listing 5.6). Since the operation returns *true*, it is guaranteed that n's state changes from "intend to insert" to "inserted" in line 34. We define the linearization point of a successful insert operation at this point. From Claim B.1 and B.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 B.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 B.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 B.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 B.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 B.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 33 holds). From Claim B.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. ■

B.1.3 Remove

Let n be the volatile node returned from the find method call in line 5 of Listing 5.7.

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 B.10, are guaranteed to be reachable and successive at its linearization point. From Claim B.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 B.7, it is not an infant. In addition, since its state is "intend to insert", from Invariant 3 of Claim B.7, it is reachable. By Definition B.5, it is not logically in the list, and by Claim B.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 B.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 B.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 B.9, it is guaranteed that the node stopped being logically in the list exactly at this step.

B.1.4 Contains

Let n be the last volatile node assigned into the `curr` variable in line 12 of Listing 5.5.

Claim B.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 B.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 B.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 B.12. From Claim B.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 B.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 B.12, in which n is reachable. From Claim B.1, n's state at this point is "intend to insert" as well and, thus, it is not logically in the list. From Claim B.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 B.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 B.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 B.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.

B.2 Durable Linearizability

As in Section A.1, we use the notion of *durable linearizability* [IMS16] for correctness. The recovery procedure, executed after a crash (and described in Section 5.4), 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 B.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 19 of Listing 5.6, 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 5.2). The `create` method can only be called from line 31 of Listing 5.6, and the `destroy` method can only be called from line 15 of Listing 5.7, 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 34 of Listing 5.6, only after the termination of the first `create` call, and is set to "inserted with intention to delete" in line 14 of Listing 5.7, before the first invocation of the `destroy` method (and by Claim B.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 B.14 (Non-Removed Persistent Nodes)**.** *Let n be a volatile node, and assume its representing persistent node has already been created in line 31 of Listing 5.6. 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 B.13, the state of n's representing persistent node becomes *valid* and not *removed* when it is created in line 31 of Listing 5.6. In addition, from Claim B.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 B.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 5.7. Then the state of its representing persistent node does not become valid and not removed anymore.*

Proof. As shown in Claim B.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.

B.2.1 Insert

Before defining the conditions for the survival of an insert operation, we need to redefine the success of an insertion in the presence of crash events.

Definition B.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 19, the `result` variable is assigned with `true` in line 27, and the respective persistent node of n is created in line 31 by some thread before any crash event.

The operation is unsuccessful if it returns `false`.

Definition B.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 B.16. In this case, its linearization point is set to be its original linearization point, presented in Section B.1.2.
2. The operation is successful according to Definition B.16, and some thread (not necessarily the one that executes the successful insertion) changes the state of the node allocated in line 19, in line 34. In this case, the linearization point is set to be its original linearization point as well.
3. The operation is successful according to Definition B.16, and no thread changes the state of the node allocated in line 19, in line 34. In this case, the linearization point is set to be the insertion of a new respective volatile node to the list during recovery.

Claim B.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 B.16). We are going to show that n is logically inserted into the volatile list at the operation's linearization point (presented in Definition B.17).

If some thread (not necessarily the one that executes the successful insertion) changes the state of n in line 34, then by Definition B.17, the operation's linearization point is this change. As proved in Section B.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 B.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 B.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 B.16, a respective persistent node of n is created in line 31. From Claim B.1, n's state does not change at all before the first crash and therefore, by Definition B.5, it is not logically in the list. As described in Section 5.4, during recovery, a new volatile node, representing n, is logically inserted into the list, and by Definition B.17, this is the linearization point of the operation in this case. Notice that by Claim B.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 B.16, it is also unsuccessful by the original definition. From Section B.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 B.14). ■

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

Proof. By Definition B.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 B.5. In addition, by Definition B.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. ■

B.2.2 Remove

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

Definition B.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 B.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 B.20. In this case, its linearization point is set to be its original linearization point, presented in Section B.1.3.
2. The operation is successful according to Definition B.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 B.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 B.22. *A surviving remove operation takes effect instantaneously at its linearization point.*

Proof. First, assume a successful remove operation (according to Definition B.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 B.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 B.21, the operation's linearization point is this change. As proved in Section B.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 B.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 B.20, the respective persistent node of n is marked as removed in line 15. From Claim B.1, n's state remains "inserted with intention to delete" until the first crash event. By Claim B.9, it is logically in the list

until this crash event. As described in Section 5.4, during recovery, a node representing n will not be inserted into the list (since its representative is marked as deleted, by Claim B.15), and in particular, will not be reachable. By Definition B.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 B.21.

When a remove operation is unsuccessful by Definition B.20, it is also unsuccessful by the original definition. From Section B.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 B.15, any persistent representative would have a *valid* and *removed* state) and thus, the unsuccessful operation indeed returns a correct answer. ■

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

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

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

B.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 B.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 B.1.4.

Claim B.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 B.1.4 ■

Claim B.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 B.1. *The* SOFT *list is durable linearizable.*

Proof. By Definition B.17, B.21 and B.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 B.18, B.22 and B.25, each operation takes effect instantaneously at its linearization point. By Claim B.19, B.23 and B.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 [IMS16]. ■

B.3 Lock-Freedom

Similarly to (and following the discussion in) Section A.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 B.27. *There is a finite number of state changes during α.*

Proof. An insert operation must terminate after executing line 34 in Listing 5.6. Likewise, a remove operation must terminate after executing line 17 in Listing 5.7). In addition, any remove operation includes at most two successful state changes (in lines 14 and 17 of Listing 5.7). Since the number of running operations is finite by assumption, the number of state changes is finite as well. ■

Claim B.28. *There is a finite number of pointer changes during α.*

Proof. The loop in lines 5–30 of Listing 5.6 must eventually terminate after a successful CAS execution in line 24 and therefore, there are no pointer updates in lines 21 and 24 of Listing 5.6. Thus, pointer updates can only occur in line 5 of Listing 5.4. When executing this update, the `pred` node is reachable from Claim B.2 and Invariants 1 and 3 of Claim B.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 B.2. *The* SOFT *list is lock-free.*

Proof. From Claims B.27 and B.28, 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 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. ■

B.4 Theoretical Bound

In this section we show that SOFT matches the lower bound presented in [CGZ18]. We need to show that both insert and remove execute at most one `psync`, and that contains executes none.

In the entire code there are two functions which explicitly execute a `psync` instruction and they are `Node::help` and `Node::destroy`.

The code for insert appears in Figure 5.6, in which we call `Node::help` outside of the main loop in line 31. So insert itself execute only one `psync`.

In remove (Figure 5.7), we call `Node::destroy` once in line 15. Again, this call is outside of any loops, so remove calls a single `psync`.

Both insert and remove use find and trim (Figure 5.4). These two functions do not use a `psync` operations at all, and thus we can conclude that insert and remove execute at most a single `psync`.

In the case of contains, the code in Figure 5.5 clearly shows that there is no `psync`, satisfying the lower bound.

Bibliography

[AB16] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.*, 10(4):409–420, November 2016.

[ALMS17] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. Forkscan: Conservative memory reclamation for modern operating systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 483–498, New York, NY, USA, 2017. ACM.

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[BDBFW19] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 253–264, New York, NY, USA, 2019. ACM.

[BGHZ16] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. ACM.

[Bro15] Trevor Alexander Brown. 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, pages 261–270, New York, NY, USA, 2015. ACM.

[CAAL19] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 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, pages 441–454, New York, NY, USA, 2019. ACM.

[CBB14] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *SIGPLAN Not.*, 49(10):433–452, October 2014.

[CCA+12] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 47(4):105–118, 2012.

[CFL17] Nachshon Cohen, Michal Friedman, and James R. Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):67:1–67:24, October 2017.

[CGZ18] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 259–269, New York, NY, USA, 2018. ACM.

[CL16] Alexei Colin and Brandon Lucia. 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, pages 514–530, New York, NY, USA, 2016. ACM.

[Coh18] Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):143, 2018.

[CP15] Nachshon Cohen and Erez Petrank. 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, pages 254–263, New York, NY, USA, 2015. ACM.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[DDGZ18] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, Boston, MA, 2018. USENIX Association.

[DGT13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 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, pages 33–48, New York, NY, USA, 2013. ACM.

[DGT15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 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, pages 631–644, New York, NY, USA, 2015. ACM.

[DHK16] Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 36–45, New York, NY, USA, 2016. ACM.

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[FHMP18] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 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, pages 28–40, New York, NY, USA, 2018. ACM.

[Fra04] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Jennifer Welch, editor, *Distributed Computing*, pages 300–314, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[HHL+06] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[IMS16] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure

model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[Int19] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 1 2019.

[JRLR15] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. Quickrecall: A hw/sw approach for computing across power cycles in transiently powered computers. *J. Emerg. Technol. Comput. Syst.*, 12(1):8:1–8:19, August 2015.

[KPS+16] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. *ACM SIGPLAN Notices*, 51(4):399–411, 2016.

[LBC+17] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[MCL17] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, October 2017.

[Mic02] Maged M. Michael. 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, pages 73–82, New York, NY, USA, 2002. ACM.

[Mic04] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[ML18] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, 2018. USENIX Association.

[NFG+13] 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. Scaling

memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[NIK+17] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

[RKCA17] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 497–514, New York, NY, USA, 2017. ACM.

[RL19] Emily Ruppel and Brandon Lucia. 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, pages 1085–1100, New York, NY, USA, 2019. ACM.

[SDUP15] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 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, pages 4:1–4:8, New York, NY, USA, 2015. ACM.

[SS06] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[WH16] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Sym-*

posium on Operating Systems Design and Implementation (OSDI 16), pages 17–32, Savannah, GA, 2016. USENIX Association.

[WJ14] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[YMP+18] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 41–53, New York, NY, USA, 2018. ACM.

[ZS15] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, May 2015.

מתח והתאוששויות. בעבודה הזו בחרנו להשתמש בתנאי הנכונות בשם "לינאריזביליות שרידה" בה בנוסף לפעולות הרגילות שיש, נוספה גם הפעולה של קריסה. תחת התנאי הזה, היסטורית חישוב היא לינאריזבילית שרידה אם היסטורית הפעולות שנוצרת אחרי הסרת כל הקריסות היא לינאריזבילית.

בעבודה זו פיתחנו שתי שיטות להפיכת מבני נתונים לשרידים ובהתאם לאותן שיטות מימשנו מספר אלגוריתמי מילון שרידים ויעילים בעבור הזיכרון הבלתי נדיף. השימוש במילונים, כשהמימוש הידוע שלהם הוא טבלת ערבול, הוא נרחב למשל באחסון של מפתחות וערכים בקהילית מסדי הנתונים. על כן נראה שמילונים שרידים יהיו בעלי חשיבות גבוהה כאשר הזיכרון הבלתי נדיף יגיע לשוק הרחב. שתי השיטות השונות שנציג כאן יוצרות את מבני הנתונים היעילים ביותר נכון לזמן כתיבת התזה הזו, ומבני נתונים אלה מניבים ביצועים טובים יותר על מערכות שמסוגלות לשרוד נפילות. בנוסף, מבני הנתונים שמימשנו הם חסרי מעצורים, תכונה המתאימה למערכת בה עשויות להיות נפילות. זאת משום שמבני נתונים חסרי מעצורים הם מדרגיים, כלומר מגיעים לנצילות גבוהה על מספר רב של מעבדים. יתרה מכך, השימוש במנעולים במערכת בה עשויות להיות נפילות, מצריך מעקב יקר על מנת שאחרי קריסה נוכל לבטל פעולות חלקיות שבוצעו בחלקים קריטיים בעקבות קריסת המערכת. בנוסף, קינון של מנעולים הופך את המעקב למסובך עוד יותר.

הרעיון המרכזי שעומד בבסיס העבודה הזו הוא ההמנעות מהצורך לכתוב בצורה ישירה מצביעים בין צמתים שונים במבני הנתונים. השימוש במצביעים במערכת ללא קריסות נועד לאפשר גישה למידע בצורה מהירה. אולם אחרי קריסה, אין צורך לגשת למפתחות בצורה מהירה. כדי להבטיח את נכונות המילון, אחרי קריסה חייבת להיות גישה אל כל המפתחות והערכים השונים, גם אם בצורה איטית. על כן, נוכל לדאוג רק לשרידות המידע שבתוך הצמתים ולא למצביעים שבין צמתים, ובצורה כזו להפחית את כמות ההדחות שנבצע. בעקבות המסקנה הזו, קראנו לשיטה הראשונה שלנו "חסר קישורים". באמצעות מידע נוסף שנשמור צומת נוכל לקבוע האם הצומת היה חלק ממבנה הנתונים לפני הנפילה או לאו. על מנת למצוא את כל הצמתים שעשויים להיות במבנה הנתונים, נקצה את כולם בתוך אזורים ייעודים שנקרא להם "אזורים שרידים".

השיטה השניה ששמו "מדהים" (מילונים עם דרך הדחה יעילה ומושלמת), שואבת את השראתה מהשיטה הראשונה, "חסר קישורים" ומהחסם התחתון על מספר פעולות ההדחה והגדר שיש לבצע על מנת שמבנה נתונים יהיה שריד וחסר מעצורים. בשיטה זו יש סיבוך נוסף על מנת להגיע לדרך ההדחה היעילה והמושלמת. כדי למנוע הדחות מיותרות יש מצבי ביניים שמיידעים חוטים מקביליים על פעולה שמתרחשת במקביל ובכך מספר חוטים עוזרים אחד לשני על מנת להשלים פעולה אחת. "מדהים" משיג את החסם התחתון תוך שמירה על ביצועים טובים הדומים לאלה של "חסר קישורים". בעבודה זו, השתמשנו בשתי השיטות על מנת לממש שלושה סוגים שונים של מילונים: רשימה מקושרת, טבלת ערבול ורשימת דילוגים. נוסף על כך, הביצועים של רשימות וטבלאות הערבול שנוצרו על ידי הפעלת שתי השיטות שלנו הושוו לביצועים של הרשימה וטבלת הערבול השרידים הטובים ביותר שהיו קיימים טרם פרסום העבודה הזו, ושמם "חסר מעקבים". השיטות שלנו הניבו הפרש בביצועים של פי שלושה, ואף יותר.

תקציר

אחת ההתפתחויות העתידיות בנושא של זיכרון דינמי בעל גישה אקראית הוא "זיכרון דינמי בלתי נדיף" (בקיצור זיכרון בלתי נדיף). המבנה של הזיכרון הזה משלב את היתרונות של זיכרון דינמי בעל גישה אקראית (בקיצור זיכרון דינמי) ושל כונן שבבי. פעולות הגישה לזיכרון הבלתי נדיף עתידות להיות זהות לאלו של הזיכרון הדינמי ואף משך הפעולה שלהן יהיה בסדר גודל זהה. בנוסף, לעומת הכונן השבבי בו ניתן לגשת למידע ברמה של בלוקים גדולים, בזיכרון הבלתי נדיף אפשר לגשת למידע ברמת הבית. בניגוד לזיכרון הדינמי, החומרה החדשה תהיה בלתי נדיפה, כלומר במקרה של נפילת מתח של המערכת ואתחול שלה, הזיכרון הבלתי נדיף ישמור על תוכנו. התוכן הנשמר ישמש לבניה מחודשת של מצב הדומה לזה שהיה לפני הנפילה כדי לאפשר את המשך הפעילות.

לעומת תוכן הזיכרון הבלתי נדיף, נראה כי האוגרים וזיכרון המטמון יוותרו נדיפים (כלומר לא ישרדו נפילת מתח). על כן, המצב של מבני נתונים הממומשים באמצעות אלגוריתמים סטנדרטים עשוי לא להיות עקבי בהסתמך רק על המידע הקיים בזיכרון הבלתי נדיף. כלומר, אחרי נפילה ועליה מחודשת, המצב של אותם מבני נתונים לא יהיה עקבי משום שכתיבות שהיו בזיכרון המטמון לאו דווקא הגיעו לזיכרון הדינמי הבלתי נדיף, ולכן לא שרדו. יתרה מזאת, כדי לשפר את הביצועים, המעבד עשוי לשנות את הסדר שבו כתיבות מגיעות מהמטמון לזיכרון הבלתי נדיף, מה שמקשה על הזיכרון הבלתי נדיף להכיל תחילית עקבית של הפעולות על מבנה הנתונים. במילים אחרות, הסדר בו ערכים נכתבים לזיכרון הבלתי נדיף, יכול להיות שונה מהסדר בו התוכנית כתבה את הערכים הללו (הסדר בו הכתיבות הגיעו למטמון). משום כך, המימוש ותנאיי הנכונות של אותם מבני נתונים צריכים להשתנות בהתאם.

כדי למצות את היכולת של הזיכרון הבלתי נדיף, יש צורך בפיתוח של אלגוריתמים חדשים, הנקראים אלגוריתמים שרידים, שיכולים להבטיח מצב עקבי בזיכרון הבלתי נדיף אחרי נפילת מתח, ובפיתוח של מנגנוני התאוששות תואמים שירוצו אחרי הנפילה. האלגוריתמים הללו צריכים לכתוב שורות מטמון בצורה יזומה לזיכרון הבלתי נדיף כדי לוודא שכתיבות חשובות הופכות לשרידות בסדר הרצוי. כתיבה יזומה כזו אפשר לעשות באמצעות פקודת מכונה קיימת שנקראת "הדחה" שכותבת בצורה ישירה שורת מטמון לזיכרון (דינמי או בלתי נדיף) והופכת את אותה שורה ללא תקפה. בדרך כלל הדחה מלווה בפעולת גדר כדי להבטיח את סיום ההדחה לפני המשך פעולת התוכנית. העלות של פעולות ההדחה והגדר גבוהות ולכן נרצה למזער את השימוש בהן כדי לשפר את הביצועים.

בדרך כלל כשדנים במבני נתונים מקביליים, תנאי הנכונות בו משתמשים הוא לינאריזביליות. נקרא להיסטורית חישוב לינאריזבילית אם התוצא של כל פעולה נראה כאילו קרה בצורה מידית בנקודה כלשהי בין תחילת הפעולה לסופה. הוצעו מספר תנאיי נכונות שונים הוצעו עבור אלגוריתמים שרידים שהרחיבו את ההגדרה המקורית של לינאריזביליות על ידי הוספת האפשרות של נפילות

המחקר בוצע בהנחייתו של פרופסור ארז פטרנק וד"ר נחשון כהן, בפקולטה למדעי המחשב.

אני מודה לטכניון ולקרן הלאומית למדע על התמיכה הכספית הנדיבה בהשתלמותי.

מילון יעיל לא-נדיף וחסר מעצורים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

יואב צוריאל

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
חשוון התש"פ חיפה נובמבר 2019

מילון יעיל לא-נדיף וחסר מעצורים

יואב צוריאל