# A Scalable Recoverable Skip List for Persistent Memory on NUMA Machines

by

Sakib Chowdhury

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Interest in recoverable, persistent-memory-resident (PMEM-resident) data structures is growing as availability of Intel Optane Data Center Persistent Memory increases. An interesting use case for in-memory, recoverable data structures is for database indexes, which need high availability and reliability. Skip lists are a data structure particularly well-suited for usage as a fully PMEM-resident index, due to their reduced amount of writes from their probabilistic balancing in comparison to other index data structures like B-trees.

The Untitled Persistent Skip List (UPSkipList) is a PMEM-resident recoverable skip list derived from Herlihy et al.'s lock-free skip list algorithm. It is developed using a new conversion technique that extends the RECIPE algorithm by Lee et al. to work on lock-free algorithms with non-blocking writes and no inherent recovery mechanism. It does this by tracking the current time period between two failures, or failure-free epoch, and recording the current epoch in nodes when they are being modified. This way, an observing thread can determine if an inconsistent node is being modified in this epoch or was being modified in a previous epoch and now is in need of recovery. The algorithm is also extended to support concurrent data node splitting to improve performance, which is easily made recoverable using the extension to RECIPE allowing detection of incomplete node splits.

UPSkipList also supports cache-efficient NUMA awareness of dynamically allocated objects using an extension to the Region-ID in Value (RIV) method by Chen et al. By using additional bits after the most significant bits in an RIV pointer to indicate the object in which the remaining bits are referenced relative to, chunks of memory can by dynamically allocated to UPSkipList from multiple shared pools without the need for fat pointers, which reduce cache efficiency by halving the number of pointers that can fit in a cache line. This combines the benefits of both the RIV method and the dynamic memory allocation method built into the Persistent Memory Development Kit (PMDK), improving both performance and practicality. Additionally, memory manually managed within a chunk using the RIV method can have its recovery after a crash deferred to the next attempted allocation by a thread sharing the ID with the thread responsible for the allocation of the memory being recovered, reducing recovery time for large pools with many threads active during the time of a crash.

Comparison was done against the BzTree of Arulraj et al., as implemented by Lersch et al., which has non-blocking, non-repairing writes implemented using the persistent multi-word CAS (`PMwCAS`) primitive by Wang et al., and a transactional recoverable skip list implemented using the PMDK. Tested with the Yahoo Cloud Serving Benchmark (YCSB),

UPSkipList achieves better performance in write-heavy workloads at high levels of concurrency than BzTree, and outperforms the PMDK-based skip list, due to the PMDK-based skip list's higher average latency. Using the extended RIV pointers to dynamically allocate memory resulted in a 40% performance increase over using the PMDK's fat pointers. The impact of NUMA awareness using multiple pools of memory compared with striping a single pool across multiple nodes was found to only be a 5.6% decrease in performance. Finally, recovery time of UPSkipList was found to be comparable to the PMDK-based skip list, and 9 times faster than BzTree with 500K descriptors in its `PMwCAS` pool.

Correctness of UPSkipList and its conversion and recovery techniques were tested using black-box recoverable linearizability analysis, which found UPSkipList to be free of strict linearizability errors across 30 trials.

**Acknowledgements**

I would like to thank Professor Golab, whose patience, guidance, and support has been monumental in my ability to write my Master's thesis. I would also like to thank the University of Waterloo ECE Department for their relentless support, and the Ontario Ministry of Universities and Colleges. I would also like to thank all of the essential workers keeping Waterloo Region, Ontario, and Canada running during this time of crisis, without whom I would not have been able to write this thesis from the safety of my home. Finally, I would like to acknowledge that this thesis work, conducted at the University of Waterloo, was done on the traditional territory of the Neutral, Anishnaabeg, and Haudenosaunee Peoples. The University of Waterloo is situated on the Haldimand Tract, the land promised to the Six Nations, which includes six miles on each side of the Grand River.

## Dedication

To everyone.

# Table of Contents

# List of Tables

# List of Figures

# List of Algorithms

# Chapter 1

# Introduction

In 2011, Marc Andreesen—software engineer, creator of Mosaic and Netscape, and modern-day investor—proclaimed that "Software is eating the world" [4]. His observance was the fact that all corporations, regardless of their primary service, are now effectively software companies. As stated by Andreesen, Netflix, Spotify, and Amazon use software to fulfill their purpose of delivering movies, music, and books (as well as movies, music, and everything else, in Amazon's case) respectively, to consumers.

On the less obvious side of things, banks, grocery stores, restaurants, libraries, manufacturers of physical goods like cars and airplanes, hospitals, and governments are all software companies now too. Whether produced in-house or contracted out, they have to make effective use of software at every level of operation to stay competitive and to provide the features and services that modern-day consumers expect. They use it to manage customer and patient information, arrange curbside pickups and contactless deliveries, catalogue inventories, analyze data, automate mechanical processes, and seemingly bend the laws of physics. This all-consuming nature of software in every aspect of our lives is not necessarily a good thing.

When software fails, bad things happen. The Boeing 737 MAX crashes and subsequent global grounding was caused by a software engineering failure [48]. A poorly-run simulation might fail to correctly predict the path of a hurricane in time to evacuate those in danger, or detect cancer in a patient. On a less life-threatening, more inconvenient level, failure at the wrong time could prevent a loved one from receiving an important message, a shopping service to lose track of your goods and money, or simply prevent a company from providing services to their customers for a number of hours. When they fail, these systems that millions interact with at light-speed often have to be fixed on much more human

timescales.

As engineers, part of our job is to reduce this recovery time as much as possible. When a system crashes or the power goes out, all data stored in volatile memory is lost, and has to be reloaded from slower secondary storage by the CPU. With the introduction of cost-effective non-volatile random access memory in the form of Intel Optane DC persistent memory, the opportunity now exists to shorten the time reloading takes, and in some applications remove the need for it entirely. The development of recoverable data structures for applications using persistent memory is an ongoing area of research. Key questions include what needs to be made persistent, how it is best persisted, and what trade-offs result from making a data structure recoverable.

The answers to these questions vary from application to application, since different data structures have different characteristics that need to be maintained for them to be useful, such as scalability on multi-core and NUMA machines or rapid recovery in highly failure-prone environments. This thesis will be investigating the right way to build a scalable recoverable skip list on NUMA machines using persistent memory.

## 1.1 Motivation

Large sets of data for many different applications are usually kept in secondary storage. Database systems use in-memory indexes, traditionally B+trees, to quickly search for desired data without having to perform slow scans of the secondary storage. These indexes are lost and have to be reloaded in the event of a crash or failure. Having to maintain recent backups for this purpose requires resources as well. Persistent memory allows these indexes to survive a power failure, and reduces the need for backing up for this purpose. Then, recovery merely consists of repairing any operations that were in progress at the time of the failure, returning the data structure to a consistent state so that operation can resume.

Skip lists are better suited to be used as a data structure for persistent memory indexes over traditional B+trees and hash tables for a variety of reasons. Balanced trees like B+trees require rebalancing to maintain their performance invariants. Skip lists, while having inferior worst-case performance, maintain the same average time complexity due to their probabilistic nature, reducing the theoretical number of memory operations necessary. This is beneficial because while comparable in performance, persistent memory has higher latency than DRAM, so the cost of every memory access is higher. Skip lists, as well as B+trees, also allow for the scanning of ranges of data falling within search limits, unlike

hash tables. It is due to these differences that skip lists become more attractive in-memory index data structure as data centres begin to use persistent memory.

For these reasons, investigating how to effectively build skip lists is an important research question, especially in this early stage of persistent memory's adoption. The insight gained, listed below in Section 1.2, is also applicable to building other recoverable data structures; skip lists were chosen due to their interesting design and immediate potential benefit to database applications.

## 1.2 Contributions

The research contributions of this thesis are as follows:

- Untitled Persistent Skip List (UPSkipList), a fully-PMEM-resident recoverable skip list written in C++ for use with applications that run on persistent memory. Its operations satisfy the correctness property of strict linearizability.
- An extension to RECIPE [47] that allows for the conversion of lock-free algorithms with non-blocking writes that do not fix inconsistencies to be recoverable in persistent memory.
- A memory management system for UPSkipList that allows the dynamic allocation of multiple segments of memory in different memory pools with objects in the segments referenced by a single word using an adaptation of the Region-ID in Value (RIV) method [15], preventing loss of cache efficiency due to fat pointers, with recovery deferred to the next attempted allocation by a thread after a crash failure
- An evaluation of the impact of NUMA-awareness using the RIV method to identify pools on different nodes in contrast to striping a single pool across multiple NUMA nodes on PMEM-resident data structure performance.

## 1.3 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 contains background information regarding persistent memory, synchronization, and data structures. Because of how recent the introduction of persistent

memory has been, it is beneficial to examine the strengths, weaknesses, and programming challenges that it presents.

- Chapter 3 is a literature review of relevant research regarding recoverable data structures, recovery techniques, and concurrent skip lists.
- Chapter 4 details the implementation of UPSkipList, the recoverable skip list presented in this thesis.
- Chapter 5 evaluates the effectiveness of UPSkipList in comparison with BzTree [5], and a transactional, lock-based skip list built using the Persistent Memory Development Kit (PMDK) [23]. Evaluation is done in terms of both runtime performance and recovery time.
- Chapter 6 details the correctness testing of UPSkipList to verify that no violations of strict linearizability can be found.
- Chapter 7 concludes the thesis.

# Chapter 2

# Background

Due to the novel nature of persistent memory, it is beneficial to first examine what it is, how it works (from a software engineering standpoint), and the techniques that have been developed to take advantage of the benefits it provides over traditional dynamic random-access memory (DRAM) and secondary storage.

## 2.1 Persistent Memory

Persistent memory (PMEM) is an emerging technology that provides byte-addressable access to non-volatile storage using the processor's load and store instructions. It does so at latencies that are low enough that the CPU can simply wait for data to be read [58].

Compared to PMEM, traditional DRAM has lower latencies, but requires regular refreshing to prevent leakage from its memory cells erasing their contents, making it volatile. This prevents data from surviving across reboots or system failures, unless it is transferred to secondary storage. Unlike PMEM, however, DRAM is effectively infinitely rewritable, while PMEM has a limited lifetime before wearing out [51].

Compared to secondary storage, PMEM offers latency that is orders of magnitude lower than spinning-platter hard drives, and until recently, flash-based solid-state drives as well [46, 49]. Regardless of latency, secondary storage accesses happen in blocks, which are usually 4096 bytes in size. While both PMEM and secondary storage can be accessed by having their contents memory-mapped so that an application in userspace can read and write their contents the same way as DRAM, the block size and high latency reduce the

throughput of secondary storage in this application by an order of magnitude, especially for random accesses [58].

### 2.1.1 Challenges

Using persistent memory presents unique challenges. Because data can now persist across runs of the program, care must be taken to ensure the in-memory data can be returned to a consistent, well-defined state in the event that an operation or program is interrupted and restarted, similar but orthogonal to ensuring consistency in multiprocessing [22]. Throughout this thesis, the process of returning data to a consistent state after a failure is referred to as recovery.

In addition, no assumptions can be made as to what state information from the CPU is available for recovery unless it is explicitly persisted to the memory, from the CPU cache. While similar to writing to disk, the lack of a kernel call and byte-level instead of block-level access makes persistence using PMEM much cheaper OS-wise and it becomes more feasible to interleave persistence operations with data manipulation to allow recovery [58]. As with all concurrent shared-memory systems, care must be taken to ensure data is persisted and becomes visible to other threads in the right order to allow algorithms to behave correctly. Memory fences and flush operations are used to prevent the processor or compiler from reordering seemingly independent operations in a single thread of execution, as required [58].

Finally, great care must be taken to prevent memory leaks and loss of ownership of regions of memory. Rebuilding data structures "from scratch" after a crash is no longer an opportunity to reclaim lost memory, and byte-level addressability means there is much less metadata assigning ownership, unless such information is explicitly persisted or tracked through the static layout of the structure itself [12]. Memory leaks in persistent memory can effectively become permanent.

### 2.1.2 Official Resources

Intel, being the primary vendor of persistent memory and the processors that can use it, has made available the Persistent Memory Development Kit (PMDK) [23] that contains numerous libraries and utilities to ease the use of PMEM in C, C++, and Java, without having to resort to inline assembly. Some of the libraries also provide solutions to the problems of recovery and memory allocation.

The PMDK website also includes documentation and informal blog posts concerning best practices, tips, and tricks when using the libraries. Libraries of note to this project in the PMDK include:

- `libpmemobj` — a "transactional object store" that handles memory allocation for objects denoted as PMEM-resident, and allows recoverability by wrapping memory modifications in transactions that can be rolled back if necessary. It internally uses `libpmem` to interface with memory.
- `libpmemobjcpp` — C++ bindings for `libpmemobj`, adding programming conveniences like "Resource Acquisition is Initialization" (RAII) and other useful idioms.
- libpmem — a library that provides a low-level platform-agnostic interface to PMEM.

The `libpmemobj` library is recommended as the starting point for developing applications that use PMEM [23], and, as explained in a later chapter, is used to establish the baseline "naive" implementation of a PMEM-resident concurrent skip list. Limitations of the naive method will become clear later in this thesis.

### 2.1.3 Performance Characteristics

The Non-Volatile Systems Laboratory at the University of California San Diego (UCSD) has had early access to Intel Optane DC Persistent Memory and has published basic performance measurements of it, compared to DRAM [46]. In their report, Izraelevitz et al. found that in "app-direct" mode, which allows the PMEM to be used as a separate persistent memory device with reads and writes done in userspace, the following relevant performance measurements of the device were obtained, listed in Section 3 of the report:

- Random read latency is on average 305 ns for PMEM, compared to 81 ns for DRAM. It is 3x slower.
- Random write latency, which they measured as temporal and non-temporal stores of up to 256 bytes and includes flushing/persisting changes to PMEM, is on average 94 ns for PMEM, compared to 86 ns for DRAM. This low latency is for when data reaches the persistent domain of the PMEM memory controller, which guarantees persistence, regardless of whether it has been physically written to the PMEM.

- Random-access bandwidth is roughly 2.8 GB/s for loads and 1.5 GB/s for stores. PMEM operations of less than 256 B, which is the device's internal block size, waste bandwidth, since the full 256 B will still be rewritten. This is despite the memory still appearing as byte-addressable to the CPU. In comparison, DDR SDRAM performs operations at 8 B "block" sizes, which is the smallest amount of memory addressable by a 64-bit processor.

The insight gained from these benchmarks is that random accesses should be done with contiguous data flushed and stored in 256 byte-aligned increments if possible, and that cache fetches should be minimized, due to the high read latency. Cache write-back latency is much more important for PMEM compared to DRAM because recoverable algorithms have to ensure certain data is persisted before proceeding, rather than relying on them being flushed when necessary. This is because a power failure will not force a flush to the persistence domain in the same way that a concurrent access from another CPU might. As such, the comparable performance for writes is promising. There is a trade-off resulting from the write latency being hidden by the PMEM memory controller's persistent domain, however. The controller has limited bandwidth relative to the processor, and saturates quickly at a low number of concurrent threads. This can be mitigated slightly by storing data simultaneously to multiple PMEM DIMMs, interleaving/striping it in a manner similar to RAID arrays.

Izraelevitz et al. also detail the performance of applications modified to store their data structures directly in PMEM, compared to persisting to disk. Depending on the effectiveness of integration with the application, improvements in performance can vary wildly, with up to 3.5x achieved with RocksDB, compared to 20% for Redis. Other applications they tested typically fell somewhere in between those two extremes. There is a clear performance benefit to switching to persistent memory to store recoverable data structures, over storing data structures in DRAM and persisting using secondary storage.

These performance characteristics have also been confirmed by researchers at the Technical University of Munich and the University of Jena [62]. They too make the recommendation to store to memory in 256-byte increments whenever possible.

### 2.1.4 Memory Model

In app-direct mode, persistent memory is accessed like a DRAM heap by a program but allows the data to outlive the program's runtime, like a file. Due to being in PMEM, the data is able to survive a power failure. The memory model adopted for the development of persistent-memory-resident recoverable applications works as follows [58]:

- The pool of memory holding the persistent heap of a program is managed as a file by the operating system. The persistent memory module itself in app-direct mode is seen by the OS as storage device that can hold these heaps and other files. It can be written to using standard file system kernel-level calls.

- To allow the program to read and write from it the same way it would from DRAM, the pool is memmapped into the program's virtual address space in a non-deterministic fashion. This means that any pointers in the pool that point to other portions of the pool will have to potentially resolve to different virtual memory addresses upon recovery.

- To allow an algorithm to recover, information will have to be persisted to the memory regularly during normal operation. This means that it has to be written from the CPU cache back to the memory itself, since the CPU cache is lost in the event of a power failure. The program uses cache line flushes to persist the information before continuing operation. Cache line writebacks have been added to the Intel x86 instruction set, allowing cache lines to be written back without evicting them from the CPU cache, removing the need to fetch them again from memory. Though the instruction can be used, this feature may not be fully implemented, with unconfirmed reports that it does not evict cache lines on Ice Lake [10].

- As mentioned before, memory fences must be used to prevent out-of-order execution from persisting modifications in the wrong order, so that concurrent accesses by other processors can be synchronized correctly.

## 2.2 Linearizability

Correctness of a data structure's implementation determines whether it reliably follows its defined behaviour and never acts in an undefined manner.

Linearizability is a correctness condition for data structures concurrently accessed by multiple processes. It removes the need to define and reason about complex concurrent behaviours by allowing operations to be treated as though they instantly occur at a single point in time, so that they can now be reasoned about as if they are done by a single process in a sequential order [42].

The point in an algorithm where the operation is treated as "taking effect" by all other operations is known as the linearization point. Strict linearizability does not allow

operations to take effect after a crash, with the crash acting as the "deadline" by which all operations must have occurred, if they occur [2].

Recovery procedures may result in operations that were in-flight at the time of the crash linearizing after post-crash operations, for example, if recovery results in the completion of a lock-free operation that swaps out a value written after the crash. Recoverable linearizability accounts for this possibility, relaxing the order in which cross-crash operations are allowed to take effect [8]. This results in well-defined behaviours that can be treated as sequential despite breaking the rules of strict linearizability.

Durable linearizability considers that real-world failures are full-system failures, and that no pre-existing threads will return to complete operations after the crash. It allows operations to take effect due to the work of any of the new threads [45], though this adds complexity to algorithm design. In applications where new threads are able to reuse the IDs of threads that existed prior to the crash, the program can treat them as pre-existing threads recovering, allowing the use of recoverable linearizability instead of durable linearizability as the algorithm's correctness condition.

## 2.3 Key-Value Stores

Key-value stores, maps, or associative arrays are a data type that associate a value with a key and store it in a set, allowing a lookup of that value using that key [44]. This is an important abstraction useful for many applications, including dictionaries and indexes.

Key-value stores usually provide the following interface [54]:

- INSERT – adds a key-value association to the set. For the purposes of this thesis, inserts may update and return an existing value associated with a key if it already exists in the set.
- REMOVE – removes a key-value association from the set. Remove operations return whether the key being removed existed in the set.
- CONTAINS – indicates whether the set contains a key. In most practical applications, this operation returns the value associated with the key as well.

An additional feature useful for many applications is a range query. It requires that the keys in the set are totally ordered, and can be defined as follows [54]:

- RANGE – return all keys in the set that fall within lower-bound and upper-bound values. In practical applications, the values associated with the range are desired as well.

For a range query to be linearizable, it must return values that all existed at the same point in time during the range query, with no additional values. This means that if a value is removed from the set after a range query has already seen it, this value must not be returned if the range query is to linearize after its removal. Likewise, values added after a range query could have seen them must be returned for range queries to be linearizable after the values' addition.

### 2.3.1 Implementation Comparison

There are several underlying data structures that are commonly used to implement a key-value store abstract data type. These include linked lists [37], skip lists [57], B+trees [19], and hash-maps [52], among others. The ones listed are the ones relevant to this thesis, and they are chosen for a specific application depending on its time complexity requirements. The time complexity, as determined for sequential implementations of these data structures, for each of the operations is as follows:

| | linked list | skip list | B+tree | hash-map |
|---|---|---|---|---|
| INSERT | $O(n)$ | $O(\log(n))$ | $O(\log(n))$ | $O(1)$ |
| REMOVE | $O(n)$ | $O(\log(n))$ | $O(\log(n))$ | $O(1)$ |
| CONTAINS | $O(n)$ | $O(\log(n))$ | $O(\log(n))$ | $O(1)$ |
| RANGE | $O(n)$ | $O(m + \log(n))$ | $O(m + \log(n))$ | $O(mn)$ |

Table 2.1: Expected time complexity. $n$ is the number of keys in the set. $m$ is the number of elements returned by a range query. Hash-maps can be augmented to have lower time complexity for range queries.

B+trees provide rapid look-ups along with linear-time range queries. Hash-maps provide constant-time look-ups, at the expense of range queries [43]. Linked lists provide linear-time look-ups and range queries, while skip lists provide rapid look-ups like B+trees and still support linear-time range queries. Skip lists have inferior worst-case performance, however, which is summarized in this table:

These worst-cases, however, are extremely unlikely, requiring the list to have degenerated into a linked list, as will be seen in studies done on skip lists in the next chapter.

| | linked list | skip list | B+tree | hash-map |
|---|---|---|---|---|
| INSERT | $O(n)$ | $O(n)$ | $O(\log(n))$ | $O(n)$ |
| REMOVE | $O(n)$ | $O(n)$ | $O(\log(n))$ | $O(n)$ |
| CONTAINS | $O(n)$ | $O(n)$ | $O(\log(n))$ | $O(n)$ |
| RANGE | $O(n)$ | $O(n+m)$ | $O(m+\log(n))$ | $O(mn)$ |

Table 2.2: Worst-case time complexity.

### 2.3.2 Skip List Operation

Skip lists, developed by William Pugh in 1989, are key-value store data structures that perform similarly to balanced trees without the need for explicit rebalancing [57]. It accomplishes this by using a multi-level structure where the probability that the next level contains a link to the desired element increases by a factor of $s$, where $s$ is the ratio between the number of elements on adjacent levels of the structure. Searches start at the highest, sparsest levels of the structure, descending to the portion of the next level that could contain the element whenever it determines the current level does not. This probabilistic behaviour, which acts like a search tree in that a roughly $(1/s)^i$ portion of the structure is narrowed down to with each traversal to the $i$-th level, does not need balancing as long as the highest level a node is inserted into is chosen randomly, maintaining that rough balance. This makes it well-suited for applications where balanced tree behaviour is desired but with minimized writes, such as a PMEM-resident index.

Internally, a skip list can be thought of as a linked list of towers of varying heights, with each level of a tower containing a pointer to the next tower that contains that level, as seen in Figure 2.1. A search operation begins from the highest level of the head sentinel node, following links on the same level until it finds a node with a value greater than the desired node. This indicates that this level does not contain the node, and that the node is contained on some lower level between the previous node and the next node on this level. The search then goes to the previous node, moves down a level, and repeats the search process.

An insert operation occurs like a search, but makes a note of the last node on each level that it encountered before moving downwards. Once the height of the new node is randomly determined, the new node is created and linked in at all levels up to its height, by first linking it to its successor node on each level. Its predecessor nodes are then modified to point to the new node, completing the insertion.

A remove operation first finds the node to be removed, and then proceeds like the inverse of an insertion. All the nodes pointing to the node being removed are modified to

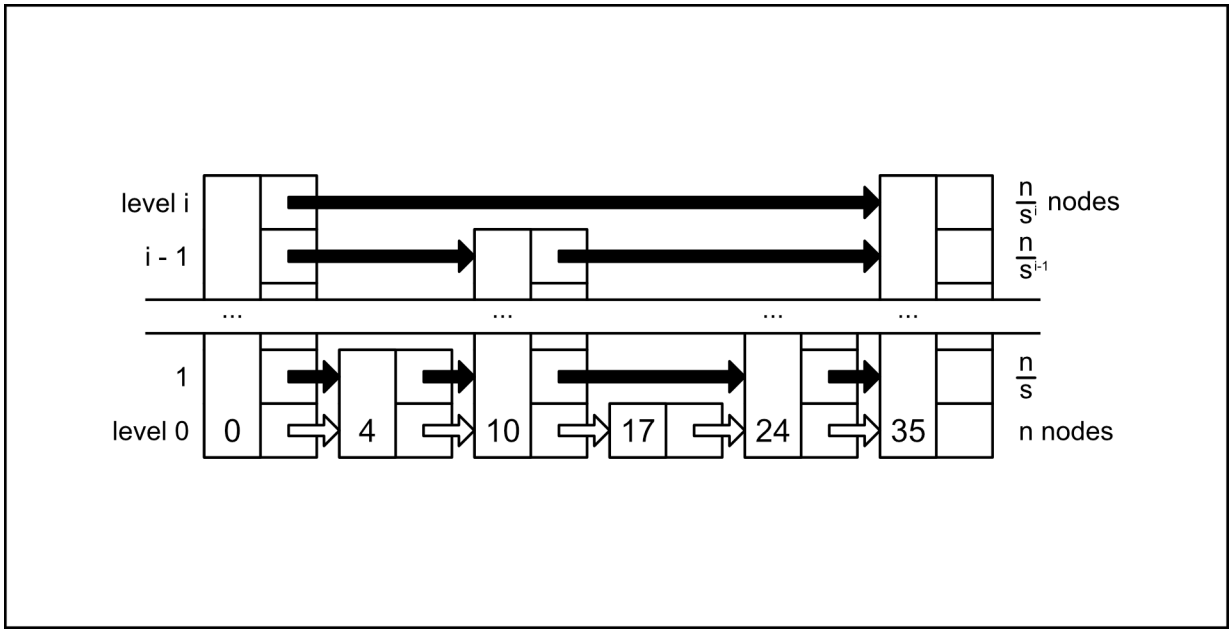


Figure 2.1: Skiplist of height $i$, with $n$ nodes on the bottom-most level.

point to the nodes that this node points to.

A range or scan operation proceeds like a search, finding the first node that falls within a range. Then, it collects all nodes on the bottom layer of the list until it reaches a node that is not within the range, at which point it processes the collection and returns.

## 2.4 Synchronization and Concurrency

Concurrent execution of operations to achieve a linearizable behaviour requires the solution of the mutual exclusion problem to prevent data race conditions or ensuring that threads can cooperate without causing nonlinearizable behaviour. This can be achieved through the use of various synchronization methods, which on shared-memory systems fall into the categories of blocking, non-blocking, or transactional methods. Unless otherwise noted, all content in this section references *The Art of Multiprocessor Programming* by Herlihy and Shavit [41].

### 2.4.1 Progress Conditions

In concurrent settings, certain guarantees of progress are given to prevent all threads concurrently accessing the same shared objects from becoming deadlocked or not making progress. Wait freedom is a non-blocking progress condition that guarantees that every thread finishes its operations in a finite number of steps. Lock freedom guarantees that at least some thread is finishing its operations in a finite number of steps; it guarantees that the system as a whole is making progress, though some threads are allowed to be starved. This makes it a weaker condition that wait freedom. Lock freedom is also a non-blocking condition.

Deadlock freedom is a progress condition that guarantees that if a thread is trying to acquire a lock, some thread will succeed in acquiring the lock. Unlike non-blocking conditions, it is also dependent on threads eventually unlocking the locks they have taken, which is necessary to prevent deadlock. Systems with locks cannot be lock- or wait-free, as a thread with a lock may block other threads without itself making progress, though they can be deadlock-free, ensuring that given enough time and fairness progress can be made.

### 2.4.2 Blocking Synchronization

Blocking synchronization involves the communication of the use of a resource between threads, with unavailability of that resource resulting in the requesting thread waiting, or being blocked, until that resource is available. Care needs to be taken to avoid deadlock, where a thread requesting multiple resources fails to acquire all of them, waiting on resources acquired by other threads that are themselves waiting on resources acquired by the first thread. Under heavy contention, most threads can end up continuously checking and waiting for resources, drastically reducing performance.

#### Semaphore

Semaphores are among the simplest of synchronization primitives, requiring a counter that signals to readers the number of rights to access to a resource that can be made. When non-zero, access is claimed by decrementing the counter, and conversely, released by incrementing the counter. When zero, a thread will wait for the counter to increase, upon which it will try to decrement it to claim the resource. In all forms of synchronization, the incrementing/decrementing of memory used for synchronization must be done atomically.

#### Mutexes

A mutex associated with a resource allows a thread to claim that resource by locking it, and release its claim by unlocking it. If a thread goes to lock a mutex that is already locked, then that thread waits for it to be unlocked before it can lock it and proceed into its critical section of code that acts on the resource. Internally, a mutex can be implemented using a binary semaphore. Unless explicitly allowed, a thread trying to lock a mutex that it has already locked results in deadlock. To prevent this, a reentrant lock can be acquired by a thread multiple times, and then released by that thread the same number of times to unlock it.

#### Reader-writer locks

Reader-writer, or shared/exclusive locks, allow synchronization of a resource that can be safely read by multiple threads but only written to by a single thread at a time. To read, a thread must acquire a read lock on the resource, incrementing the number of readers of the resource, which can only be done if there are no writers who have locked it exclusively, in which case it waits for the resource's write lock to be released. Once finished reading, the reader releases its read lock and decreases the number of readers. A writer acquiring access to a resource waits until there are no other writers having locked it, after which it acquires the lock. It then waits for all readers to release the lock, during which time no new readers that know of the writer can acquire access.

#### Trylocks

Trylocks blur the line between blocking and non-blocking synchronization slightly, by allowing a thread to continue execution rather than waiting on a resource if it finds it to be locked. However, they do not allow a thread to enter its critical section to manipulate that resource, and simply let a thread perform other actions on other resources instead.

### 2.4.3 Non-blocking Synchronization

Non-blocking synchronization methods do not wait for a resource to become available, and allow optimistic attempts to be made to access the resource instead. Hardware synchronization primitives, that allow atomically modifying a memory address are necessary to do so successfully. The compare-and-swap, or CAS primitive is an essential primitive

that allows a thread to read and modify a value in a memory address, and then write it back provided that the current value in that address has not changed. If it has changed, then the thread can try again, re-reading the value and attempting to change it until it succeeds. Care still needs to be taken in algorithm design to ensure progress to prevent operations requiring multiple CAS procedures from blocking each other, though greater interleaving of operations is now possible. In addition, an algorithm can be made wait-free by decoupling the progress of operations from the progress of the threads themselves, by having any thread that comes across an in-progress operation help it to completion before starting its own operation. In this way, non-blocking synchronization can avoid the pitfalls of blocking synchronization, although at lower levels of contention it tends to result in reduced performance due to increased overhead.

### 2.4.4 Transactions

Transactions offer an alternative way of thinking about operations. Rather than having a thread enter a critical section, a thread will attempt to make changes to multiple memory addresses until it succeeds, in which case the transaction commits, or fails, in which case the transaction rolls back and the original values are restored. When two threads attempt to perform transactions involving overlapping memory addresses, they conflict with each other and cause either or both transactions to fail, requiring them to restart. Transactions can be implemented in software using the aforementioned synchronization methods, or in hardware, where the processor tracks which addresses are being accessed and automatically aborts a transaction when a conflict occurs. Hardware transactions, while more performant than software transactions, are difficult to use with persistent memory, as in practice the flushing of a change from CPU caches to the persistent domain causes transactions to abort [25].

## 2.5 Memory Management

In low-level programming languages like C++, dynamic memory management has to be performed to allocate and deallocate memory to allow objects to survive beyond the scope in which they are originally created [20]. Allocation claims a position on the heap for a thread to write an object to, while deallocation returns memory used by an object to the heap so that it may be reallocated. Dynamic memory management is important for data structures that can hold an arbitrary amount of data, with new nodes in the structure being allocated from the heap and linked to the structure. While allocation has to be done

explicitly, deallocation once all references to an object have been removed can be managed automatically using various reclamation schemes [1].

### 2.5.1 Memory Allocation

Requesting memory from the allocator has to be done explicitly, with a `new` operator or `malloc` function call. Internally, the allocator manages a portion of memory known as the heap from which it removes the desired amount of memory and returns a pointer to it to the requesting program. In a multithreaded context, care needs to be taken to ensure that the allocator can safely allocate to multiple threads at the same time. The allocator's main objective in how it manages memory are to balance the overhead of managing the memory with preventing fragmentation. General-purpose allocators must be able to allocate on the order of both bytes and gigabytes. For applications where there are going to be many allocations of only a few hundred bytes at a time, optimizations can be made to reduce the overhead of general allocation by requesting a large chunk of memory and managing allocations to it manually [61]. Common methods of doing this include the following:

- Linearly allocating memory with no deallocations by simply incrementing a pointer to the next free memory address by the amount being allocated
- Including a header before each chunk of allocated memory indicating the size of the allocation, to be used during deallocation to move back the pointer to the next free memory address
- Dividing memory into equal-sized blocks linked together as a list
- Dividing memory into contiguous regions of free space linked together as a list, starting off as a single region

While all these methods offer reduced time to allocate over `malloc`, they require manually keeping track of free memory, increasing program complexity. However, when programming without the availability of `malloc` as is done in persistent memory programming, knowledge of these techniques can be valuable to improve performance over the fat-pointer-based allocator in the PMDK. This is detailed in the next chapter, in Section 3.1.

### 2.5.2 Memory Reclamation

Reclaiming memory is easy in non-recoverable, lock-based data structures where locks to all nodes pointing to the node to be removed can be acquired, their pointers removed, and the

node deallocated. For lock-free structures, and recoverable structures, memory reclamation is more difficult because pointers to a node to be deallocated have to be changed without locks, and references to the node may still be kept by other threads or recorded as part of recovery information that will be used after a crash. Multiple methods of ensuring that reachable memory is never reclaimed include:

- **Tombstoning** simply does not deallocate nodes to be deleted, and instead sets their value to a "tombstone" indicating that the node logically does not exist and should be ignored. This sidesteps the issue of garbage collection and removes it from the scope of the problem, at the expense of never decreasing memory use by the program, making it less suitable for production-ready applications.
- **Reference counting** tracks the number of references to an object, and upon the number reaching zero, the object is allowed to be deleted. There is difficulty with this technique in a recoverable context, however, as a reference count incremented prior to the addition of a reference may be interrupted by a crash and left off by one, preventing the object from being deallocated until repaired.
- **Hazard pointers** let threads indicate to the garbage collector that it has a reference to an object, preventing that object from being deallocated until all threads relinquish their references [53]. Hazard pointers have limitations that prevent them from being useful for recoverable data structures, due to difficulty efficiently determining whether no references to objects by other objects exist [11].
- **Epoch-based reclamation**, has threads indicate which epoch it removed an object in to the reclaimer. Once all threads have moved on to a future epoch in which the reclaimer can be sure no references exist to objects removed in this epoch, all these objects can be removed [11].

# Chapter 3

# Literature Review

This chapter presents a literature review on concurrent skip lists, persistent-memory-resident data structures and key-value stores, and techniques for building high-performance concurrent skip lists.

## 3.1 PMEM Programming Techniques

Many different techniques have been developed in recent years to create recoverable data structures, which a program can recover to a consistent state after a crash failure, given that some or all of the data structure is stored in persistent memory. Different techniques offer different tradeoffs between time required for recovery and overhead during runtime. The ideal technique has minimal recovery time after a crash, and minimal hit to performance during runtime, compared to a non-recoverable implementation of a data structure.

Wang et al. created the Persistent Multi-word CAS (PMwCAS) library [63] that implements Harris' CASN operation [38]. The PMwCAS operation atomically changes multiple addresses if they all contain expected values in a persistent memory environment while preventing inconsistency in the event of a crash. It facilitates the conversion of lock-free algorithms that already use CASN, also known as multi-word CAS (MwCAS), to be recoverable when run on persistent memory.

MwCAS works by installing pointers to per-operation descriptors to all relevant memory locations using CAS prior to writing the new values. MwCAS descriptors are structures containing, for each location to be changed by the MwCAS, the expected value and new value, along with metadata indicating the status of the operation. If any of these CAS

operations fail, the pointers that were successfully installed are replaced with the original values backed up in the descriptor. If they all succeed, they are then changed from the descriptor-pointer value to the expected value, preventing the intermediate inconsistent state from being visible. PMwCAS makes multiple changes to MwCAS to make it recoverable:

- One bit of a pointer is used to indicate whether it has been persisted using a flush operation. Upon reading, if this bit is set, the reader will flush the cache line containing the pointer, and then unset the bit. This ensures that any writes dependent on reads will not be persisted prior to those reads, preventing the read from being lost. Once the bit is unset, the pointer will be flushed again eventually once its cache line is evicted for any reason, with a crash prior to this simply meaning that the pointer will be flushed and the bit unset again. The correct value will be read after a crash regardless.
- A recovery function is implemented that returns the memory to a consistent state after a failure, by completing CAS operations where all descriptor-pointers were installed successfully, and rolling back CAS operations otherwise. A program calls this function prior to resuming execution after a crash. The recovery process is performed sequentially.
- Recoverable memory allocation is handled by an external allocator, while memory allocated to PMwCAS but not yet reachable within the datastructure using PMwCAS is kept track of using descriptors.
- Memory reclamation is handled using an epoch-based algorithm, and the data structure using PMwCAS to make memory unreachable can use metadata within a descriptor to signal to PMwCAS to return memory to the allocator upon success or failure. It is assumed by PMwCAS that memory marked as unreachable upon success/failure of a PMwCAS operation will not be reachable via unchanged pointers in the data structure.

This conversion shows an overhead of only 4-6% in realistic workloads.

The main limitation of the research by Wang et al. is that it was conducted prior to widespread hardware availability of Intel Optane memory. This means that their experimental results for recoverability are done using flash-backed NVRAM with the same latency characteristics as DRAM. Additionally, their recovery method does not allow operations to be processed while recovery is ongoing, and converted algorithms using multiple single-word CAS operations may still leave memory in an inconsistent state after a failure.

Lee et al. have also developed a technique for converting existing DRAM lock-free algorithms to be recoverable from PMEM, known as RECIPE [47]. Their method works on the insight that many lock-free algorithms already have methods of recovering an inconsistent state, for example due to an unsuccessful/restarted operation, to a consistent state. These methods can often be used in the event of a failure to recover in-PMEM data structures to a consistent state as well, so that execution can continue.

Lee et al. measured the performance of data structures using their techniques using the Yahoo Cloud Services Benchmark (YCSB), which generates workloads resembling realistic use cases in terms of distribution, key-value contents, and read-write ratio [24]. They found that their data structures achieved up 1.6x the performance for integer keys of FAST & FAIR, a B+ tree algorithm made for persistent memory, and up to 5x the performance for string keys. They determine this is due to better cache efficiency and reduced number of flushes in the converted algorithms. They believe the maturity of the algorithms being converted contributes to their better performance, compared to the newly-developed FAST & FAIR algorithm.

The limitation of their method is that it only applies to structures that meet any of the following criteria:

- Updates are performed by a single atomic store, and reads are non-blocking. This does not apply to skip lists, since insertions require the modification of several pointers, in order to maintain the skip list property.
- Writing operations fix inconsistencies when they are found, and both reads and writes are non-blocking. This may apply to skip lists, depending on the algorithm used.
- Writing operations do not fix inconsistencies, and reads and writes must be non-blocking and blocking, respectively. This may also apply to skip lists, depending on the algorithm used.

Intel's Persistent Memory Development Kit (PMDK) provides a transactional object store known as `libpmemobj` that can be used to convert existing algorithms to be recoverable [22]. It works by following the same model as database software transactions to achieve recoverability. Prior to modifying a memory range, a copy is made and stored in a separate location. This way, if a failure occurs and a transaction that has not committed was in progress, the original values can be restored, providing failure atomicity. Using `libpmemobj` makes the creation and conversion of applications to recoverability very easy; portions of the code that could leave the data structure in an inconsistent state if interrupted by a crash merely have to be wrapped in a transaction. The transactions provided

by the PMDK do not prevent their memory ranges from being read before they commit, however, so additional synchronization methods must be used if the application will also be concurrent.

There are additional issues that this library presents when it comes to building high-performance concurrent recoverable data structures:

- The library uses "fat pointers" to reference data stored in persistent memory. One word is used to indicate which persistent memory pool contains the data pointed to, and another is used to indicate the offset of the data from the start of the pool. This reduces the number of pointers that can fit in a single cache line, increasing misses and decreasing performance.
- There is additional overhead due to the need to copy all changed values prior to modification. Although this is a general method that will work in most cases, this write-amplification uses precious bandwidth and reduces the maximum theoretical throughput below algorithms that do not use this method.

Due to its ease of implementation, a `libpmemobj`-based skip list is used as the baseline recoverable skip list comparison, since it will be the method used by most developers getting started building recoverable data structures.

Lersch et al. have conducted research evaluating B+tree-based recoverable indexes built using various methods [50] on real hardware. This provides a good comparison as to what works well and what does not. They have developed a benchmarking framework to evaluate persistent memory indexes as well, whose design influenced the benchmarking done in this thesis. The insight they gained through their experimentation agrees with that laid out in the background section, namely that bandwidth is scarce and correctness is difficult. The structures they compare include the following:

- wBTree, whose design is optimized to minimize cache line flushes and writes [17]. They do this by leaving nodes unsorted but reduce the performance impact of this by using an indirection array to guide searches.
- NV-Tree, which enforces the consistency of leaf nodes but relaxes it for inner nodes by not maintaining recovery information for them [64]. Upon recovery, leaf nodes are consistent, and are used to rebuild the inner nodes.
- BzTree, which uses PMwCAS [63] to manage its data [5].

- FPTree, which stores inner nodes in DRAM, whose contents has to be rebuilt upon recovery.

Their experiments show that FPTree and NV-Tree tend to have the best performance, due to not enforcing PMEM consistency for inner nodes. wBTree tends to perform better in single-threaded workloads than BzTree, due to its algorithm being designed specifically to minimize flushes, instead of merely making the algorithm recoverable. wBTree does not support multithreaded operation, however. Additionally, both FPTree and NV-Tree have non-constant recovery time, unlike BzTree and wBTree.

Lepers et al. have developed KVell, a fast persistent key-value store for block devices [49]. Though it does not operate on persistent memory, instead favouring fast block devices like NVMe SSDs and Optane block devices, it provides insight into the modifications that can be made to improve performance when moving from slower secondary storage to faster secondary storage and in-between technology like PMEM. It also serves as a point of comparison in investigating whether PMEM-resident indexes are worth the effort for recoverability over block devices with comparable bandwidth.

Lepers et al. find that the CPU is the bottleneck for LSM key-value stores and B-tree key-value stores. The key method they use to improve performance is to reduce cross-core communication and synchronization, avoid sorting data when unnecessary, and avoiding syscalls and unnecessary IO operations. Using PMEM in app-direct mode in place of block devices will avoid syscalls, but algorithm design is still important to minimize processor overhead and unnecessary IO operations. Some methods of doing this include the indirection array method from wBTree [17], as previously mentioned.

David et al. have identified several techniques to implement recoverable lock-free concurrent data structures on PMEM in a log-free manner [28], similar to RECIPE [47] and PMwCAS [63]. This avoids the overhead of logging required for transactions, as in `libpmemobj` [22], making it the first option to consider to improve performance over a `libpmemobj`-based baseline implementation. As done in the RECIPE paper, David et al. focus on lock-free data structures because they always keep the structure in a consistent state and so do not inherently require logging.

The techniques they propose are as follows:

- link-and-persist – This method involves atomically changing and persisting a link, using pointer tagging to fold a validity indicator [37] into the pointer, the same as in PMwCAS [63].

- link cache – In this method, cache write-backs are batched together until they have to be written for correctness. This reduces overhead by removing per-write latency for writes that do not have to be immediately written by reducing the number of fences necessary.

- NV-epochs – This is an epoch-based memory management scheme which, like PMwCAS, interfaces with an external persistent memory allocator, albeit with some non-standard modifications. Unlike PMwCAS, which uses its descriptors to track and recover unlinked memory owned by the data structure, NV-epochs uses a page table to mark memory pages that may contain unlinked nodes, and verifies during recovery if all nodes in that table are linked. Unreachable nodes are reclaimed.

David et al. also provide recoverable implementations of many data structures using their techniques, including skip lists [29]. Their implementations are done using their custom version of `jemalloc` designed to simulate persistent memory performance in volatile memory, with the modifications required for NV-epochs. Unfortunately, this means that testing them on actual persistent memory is more difficult, requiring modifications to the external persistent memory allocators.

## 3.2 Concurrent Skip Lists

There are many different ways of making skip lists safe under concurrent accesses, with some ways having more potential in recoverable environments than others.

William Pugh proposed the first methods of allowing concurrent maintenance of skip lists [56]. His method assumes that single-pointer writes are atomic, allowing reads to occur concurrently to writes with no need for locks. Insertions and removals obtain locks on all nodes to be modified prior to the modification. Inserted nodes cannot lead an optimistic search off the list, so no special synchronization is needed there. Removed nodes, however, can. Pugh avoids this issue by modifying nodes being removed to point back to their predecessors on each level, such that a search will step back onto the previous node and follow it to the next node that it was pointing at, forcing it to linearize after the removal. The removal has to be done level-by-level until the height of a node is reduced to 1.

Herlihy et al. created a simpler lock-based skip list technique known as the lazy-skip-list [40]. Without using pointer reversal, Herlihy's skip list is easier to implement and prove correct. This is achieved using the same technique as lazy-list algorithm of Heller et al. [39], where a node is marked as logically deleted with an atomic operation, serving as the

linearization point of the deletion. Due to the multi-layer nature, an additional flag is used to indicate when a node is valid, or "fully-linked", upon insertion. Searches are done optimistically, and can continue past marked nodes without worrying about being lead off the list. This simplifies the ability to reason about and prove the list to be correct, and has resulted in Herlihy's lazy-skip-list algorithm becoming the starting point for most modern lock-based skip list algorithms.

A highly popular lock-free skip list algorithm developed by Doug Lea is included in the Java Concurrency Package [31]. It uses a technique developed by Fraser [34]. It works by CASing pointers from a node to be deleted to a marker node, which serves to prevent the former from being modified until it is physically removed from the list. To check if a node is marked, a read simply checks the next node in the list.

Herlihy et al. have also developed a lock-free skip list algorithm [41], similar to Lea's [31], based on Fraser's [34]. Instead of CASing pointers to a marker node to indicate logical deletion, their algorithm uses an abstract AtomicMarkableReference object that supports atomically changing its mark along with the address that it points at. This may internally be implemented in various ways, including using pointer tagging, or a double-compare-and-swap primitive if available. Elements are considered to be in the abstract set only if they are linked at the bottom-most level, and logically-deleted nodes are removed by searches as they traverse the list. This means that a node may be removed from lower levels by a search while an insert is still linking the node with higher levels; as a result, both Herlihy's and Lea's lock-free implementations transiently sacrifice the skip list property that higher levels of the list are always contained within lower levels.

Fomitchev and Ruppert proposed an additional technique to improve the performance of lock-free linked lists and skip lists [32]. Like Pugh [56], they use backlinks to lead a search operation back onto the list when a node is being deleted, but in a lock-free context.

Regarding best practices for building performant concurrent and non-concurrent skip lists, an article of note comes from the personal blog of open-source developer "Ticki" [60]. Although skip lists are quite simple, implementations with some poor design practices can result in dramatically decreased performance in modern systems due to cache thrashing. Ticki makes the observation that unless all levels of a tower are included in a single contiguous region of memory, cache misses will occur while the processor fetches the next level of the tower from elsewhere, instead of having it preloaded automatically. They make additional suggestions to improve performance, as follows:

- Cache efficiency can be improved further by allocating memory pools for nodes, instead of allocating for each node directly. Allocation of individual nodes is done

from the pool, ensuring that multiple nodes reside in a single page of memory that can be accessed faster than multiple pages.

- Nodes can contain multiple values; nodes of height 1 can be discarded and have their value added to the previous node, reducing the data fetched for height-1 nodes to just their value.
- Height generation can be done more deterministically by tracking the frequency of each possible height, and creating a new node with the height that has the lowest frequency relative to how many nodes of that height are desired.

Although Ticki's suggestions are for a single-threaded skip list, they can be implemented for a concurrent skip list as well. Containing multiple values in a node requires adding support for concurrent node splitting when a node becomes full.

Crain et al. have developed a lock-free skip list algorithm that reduces contention by having a small number of background threads perform the construction of the towers, while allowing normal insertion operations to return as soon as a node is linked at the bottom, becoming a part of the abstract set [26]. This serves to reduce hot-spots of attempted modifications to the upper levels of the list when multiple insertions are ongoing, preventing them from being invalidated and having to retry.

Dick et al. have introduced a skip list with a rotating mechanism that replaces towers with wheels whose size is modified in constant time to deterministically preserve the logarithmic time complexity [30]. This is similar to the rebalancing mechanism used to reduce the height of balanced search trees [7], using a skip list structure instead of a tree to avoid contention on the tree root that reducing the height of a regular balanced search tree would have. This is a novel data structure that is not quite a skip list.

Daly et al. have built a skip list algorithm called NUMASK which is optimized for non-uniform-access memory systems [27]. Their design not only mitigates the increased latency of non-NUMA-local accesses, but manages to outperform SMP implementations by taking advantage of the NUMA architecture. Their algorithm applies a modification to an existing algorithm, in this case Crain et al.'s skip list [26] to make it NUMA-aware. They duplicate the upper levels across NUMA nodes, having a per-node background thread build up the navigation structure locally. This design is also amenable to a hybrid PMEM-DRAM architecture, with copies of upper levels built up locally in the DRAM of each NUMA node.

Chen et al. have developed a recoverable skip list called NV-Skiplist [16]. Their algorithm uses a hybrid design with the bottom-most level stored in PMEM while higher

levels are stored in DRAM. While this speeds up performance, it increases recovery time, due to having to rebuild the upper levels upon recovery. Additionally, their skip list was not tested on real PMEM, due to its lack of availability when their work was published. Recoverable skip lists have also been built using the previously mentioned PMwCAS [63] and by David et al. using their log-free techniques [28], though both were also developed prior to the availability of real PMEM.

## 3.3 Memory Management

Recoverable memory management requires that interrupted allocations and deallocations can be detected and resolved after a crash failure, preventing memory leaks. The previously mentioned `libpmemobj` library offers recoverable dynamic memory allocation and deallocation, either within its transactions or using its atomic API [22] and returns the fat pointers discussed in Section 3.1. The transactional API manages interrupted allocations the same way as all other interrupted memory modifications, by rolling them back upon recovery, and is used like `malloc`. Atomic allocations using `libpmemobj` in C require the user to provide a constructor function so that the allocator can either fully initialize the object or reclaim memory after a crash. The C++ API provides the same interface for both transactional and atomic allocations, as C++ objects have constructors already that can be passed to both. Deallocations require explicitly calling destructors prior to deallocating memory. Internally, allocated objects in `libpmemobj` are stored in a linked data structure that can be iterated over, facilitating recovery if a failure occurs after an atomic allocation but before the new object has been made reachable from the root. This of course adds an $O(n)$ step to the recovery process, where $n$ is the number of elements in the allocated object linked list.

NV-Heaps is a memory allocator that, like `libpmemobj`, provides recoverable allocation through the use of a transactional allocation system. Unlike `libpmemobj` however, it only allows referencing objects stored in the same storage pool, preventing the use of NUMA-aware algorithms with multiple pools on different NUMA nodes.

For referencing memory locations in persistent memory, an alternative to fat pointers is provided by Chen et al. [15]. By using the unused, most significant bits in a pointer on 64-bit systems, where only 48 bits are usable to address memory, they store the ID of the region in which the data the pointer is referencing resides. This allows the use of multiple regions without using fat pointers, opening up the ability to use multiple NUMA nodes while maintaining the cache efficiency of smaller pointers.

Makalu is a memory allocator developed for recoverable allocation by Bhandari et al. [9]. It has garbage collection and allows usage with simple analogues of `malloc` and `free` calls. While very flexible and performant, its only limitation is that its recovery time is dependent on the size of the heap being managed.

# Chapter 4

# Implementation

This chapter describes the design of UPSkipList and the considerations and tradeoffs that went into its development. It describes the extension made to RECIPE [47] to convert Herlihy et al.'s skip list algorithm [41], the mechanism used to defer recovery of failed memory allocations, and the memory management techniques used to enable NUMA awareness without reducing cache efficiency. Testing of UPSkipList's correctness is deferred for Chapter 6.

## 4.1 Design Overview

The UPSkipList algorithm uses as a starting point the lock-free skip list design by Herlihy et al. [41], which implements a concurrent and lock-free version of Pugh's skip list [57]. The traversal operation in Herlihy et al.'s skip list algorithm is wait-free, always making progress, while inserts and removals are lock-free. UPSkipList makes modifications to Herlihy et al.'s algorithm not only to achieve the goals for a recoverable skip list as specified below, but also to increase performance by allowing the storing of multiple keys in a single node through the implementation of recoverable concurrent node splits. Due to the addition of a lock for the purpose of node splits, insertions, updates, and removes are no longer lock-free, but are deadlock-free, meaning that they cannot all block each other from performing their operation. The effects of the modifications on the progress conditions are specified in their respective sections., with deadlock-freedom ultimately being maintained by UPSkipList. Skip lists storing multiple keys have previously been developed by Platz et al. [55] and by Hillel et al. [6], though neither are recoverable. Although the original algorithm is written in Java, UPSkipList is implemented in C++

due to low-level persistence libraries in the PMDK being written in C [23]. The use of C++ allows for better examination of recoverable memory allocation techniques for lock-free data structures, with the goals of practicality for both development and use, and increasing performance, as defined below.

**Practicality Requirements** Practicality in this case is considered in terms of three criteria:

1. The usability of the data structure in a program that may have many such data structures stored within each pool of memory, regardless of the PMEM allocation system used by that pool, with potentially multiple pools of memory
2. The ability to have the data structure memory-mapped in write mode to different locations in multiple processes' virtual memory at the same time, which is possible in Linux
3. The ability to recover in constant time with respect to the size of the structure after a failure, ensuring that uptime can be maintained in a failure-prone environment regardless of the size of the data structure

The C++ memory model does not currently recognize the existence of non-serialized data structures continuing past the runtime of a program [23]. As a result, numerous recoverable data structures have been developed that use different techniques to manage memory [15, 23, 12, 63, 9, 18], described in detail in Section 3.3.

### 4.1.1 Making a Lock-Free Skip List Recoverable

Multiple modifications had to be made to Herlihy et al.'s skip list to make it recoverable, increase performance, and to work in persistent memory while meeting the practicality requirements from Section 4.1.

Directly within the algorithm, cache flushes and memory fences have to be added after writes to ensure that data has left the volatile domain and will survive power loss. This is abstracted as the "persist" primitive, shown in Function 1. These changes do not affect progress. Throughout this thesis, mutable objects are denoted using sans-serif text, while immutable values are italicized. Additionally, := is used for assignment while = is reserved for equality tests. Prior to the Intel Ice Lake architecture, the `CLWB` instruction, which flushes and does not evict cache lines, is not implemented. `CLWB` is still able to be used in

**Function 1:** Persist(Array<Address> *memoryAddresses*)

**Result:** Persistence of cache lines covering *memoryAddresses* to memory outside of cache as side effect

```
1  for cache lines covering addresses in memoryAddresses do
       /* cache lines flushed within Persist() have no specified order;
          use multiple Persist() calls to order them explicitly        */
2      CLWB(cache line)
3  end
4  MemoryFence()
```

**Function 2:** CAS(Address *memoryAddress*, uint64 *oldValue*, uint64 *newValue*)

**Output:** *True* on successful swap, *False* if expected value is not found

```
5  atomic
6      if memory[memoryAddress] = oldValue then
7          memory[memoryAddress] := newValue
8          return True
9      else
10         return False
11     end
12 end
```

Function 1, however, as it is treated as `CLFLUSHOPT` by pre-Ice Lake architectures, which flushes and evicts cache lines, adding latency to a future read of that cache line but still working correctly.

In the PMDK on x86 processors, the memory fence is implemented as an `SFENCE` instruction, which orders stores and flushes [23]. Unlike PMwCAS [63] and the log-free techniques of David et al. [28], but like in RECIPE [47] and the PMDK [23], no marking is used to indicate durability of a pointer to the reader, relying solely on the flush and fence by the writer prior to dependent writes. The RECIPE paper showed that this is sufficient to achieve the matching of store order between the CPU and persistent memory, which they confirmed by tracking cache line flushes.

The recoverable conversion of Herlihy et al.'s lock-free skip list is built upon the "persist" primitive, as mentioned, and the "compare-and-swap (CAS)" read-modify-write primitive, shown in Function 2. `CAS` is extensively used, not just in the original lock-free algorithm and its conversion, but also in the recoverable fine-grained memory allocator,

detailed in Section 4.3.3.

The addition of `Persist` at key points in the algorithm is not enough to achieve recoverability for the data structure. Interruption due to a crash failure during write or `CAS` operations between `Persist` calls can leave data in an inconsistent state. A crash immediately after `Persist` is called can also result in not all addresses being persisted from being flushed out of the volatile domain, also resulting in an inconsistent state. In both cases, repair of an inconsistent data structure is done in different ways depending on how the inconsistency can be detected.

A failure during allocation, deletion, and prior to the addition of a node to the abstract set leaves memory unreachable to both the data structure and the allocator, resulting in the leakage of memory. An example of this is shown in Figure 4.1. To prevent this leakage, a logging technique, explained in Section 4.1.4, is used to allow runtime detection of inconsistencies unreachable during traversal of the data structure. This method requires the addition of a single cache flush prior to attempting the modification during failure-free operation.

For inconsistencies that can be detected during traversal, added recovery methods are used to restore invariants and consistency of the data structure, such as the average search time complexity, finishing the splitting of a node, or to clean up links that have been abandoned by a crashed process. Figure 4.2 shows an inconsistency possible in UPSkipList that can be detected during traversal, where the new node had not been fully linked when the crash occurred, but is reachable as part of the abstract set. The next pointers from its predecessors will continue to point to its successors on higher levels instead of eventually being linked with the new node, which will not occur now without intervention.

The technique used to recover from these inconsistencies is explained in the next Sections 4.1.2 and 4.1.3.

### 4.1.2 Limitations of RECIPE and NVTraverse Conversion Techniques

The current state-of-the-art methods of converting lock-free algorithms to be recoverable, RECIPE [47] and NVTraverse [35], have several limitations that prevent them from applying directly to Herlihy et al.'s lock-free skip list algorithm to achieve the requirements from Section 4.1. They also do not deal with recoverable memory allocation or persistent pointers, which UPSkipList does, through the logging method (Section 4.1.4) and NUMA-aware offset-based pointers (Section 4.3.1). While these methods and UPSkipList can support

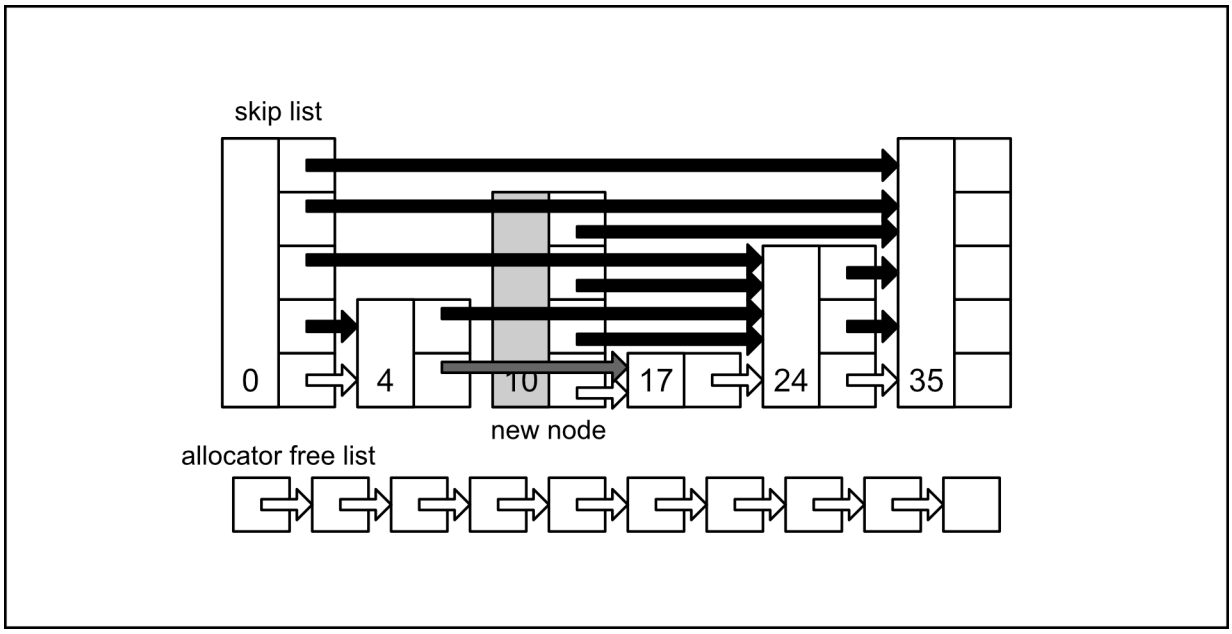


Figure 4.1: Fault possible due to interrupted `Insert` operation that cannot be detected during traversal. New node with key of "10" has not been linked in to the abstract set, but has already been removed from the list of allocatable space, leaving its memory unreachable and leaked.

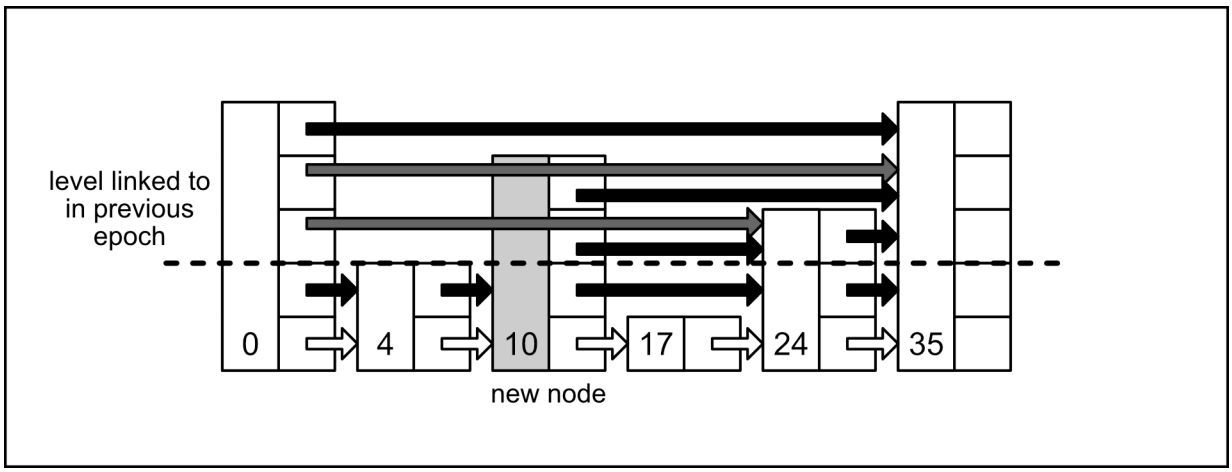


Figure 4.2: Fault possible due to interrupted `Insert` operation that can be detected during traversal. New node with key of "10" has been linked in to the abstract set (white arrows) at level 0 and at level 1, but because of the failure will not be reachable on higher levels.

NUMA machines by running on PMEM devices striped across multiple DIMMS on separate NUMA nodes, NUMA awareness using multiple pools allows for the implementation of algorithms with NUMA-aware optimizations, like in [27], [59], and [13]. The conversion method used for UPSkipList, while not automatic, fills the gaps left by RECIPE and NVTraverse.

RECIPE requires that writes in a lock-free algorithm satisfy at least one of three properties [47], and prescribes a conversion method for the operations satisfying that property. In a skip list, the write operations that need to be converted are updates, which change the value associated with an existing key, and inserts, which insert a new key-value pair into the structure. The first property states that modifications are applied using a single atomic store to perform an update. Though this situation is trivial to make recoverable, and applies to updates, insertions in Herlihy et al.'s lock-free skip list do not involve just a single atomic store, and instead require the use of multiple atomic steps to perform the operation.

The second and third properties require that writes are either non-blocking and repair inconsistencies (i.e., help finish incomplete operations), or blocking and non-repairing. Neither case applies to Herlihy et al.'s skip list algorithm. Writes do not fix inconsistencies of insertions when they are found; they simply assume that another thread is taking care of it. Reads only fix inconsistencies as part of the algorithm when removing links to logically-deleted nodes. Writes are non-blocking as well, which prevents detection of inconsistent nodes in need of repair by checking whether the nodes are locked, leaving RECIPE's third conversion method unusable for this algorithm.

To apply RECIPE's third conversion technique, it is necessary to find a way to detect when an inconsistency is due to a crash failure without having to check a DRAM-resident lock. Although volatile locks can be added to Herlihy et al.'s algorithm for this purpose, this would require that during recovery these locks be reinitialized and reassociated with the nodes, making recovery time dependent on the size of the structure. The solution to this problem is accomplished in UPSkipList using the technique described in Section 4.1.3.

Like RECIPE, NVTraverse applies to a class of algorithms, in this case formalized as "traversal data structures". In their technique, a skip list only counts as a traversal data structure if the bottom-most linked list is considered the data structure, with all upper layers treated as DRAM-resident auxillary entry points. This precludes the usage of their technique for non-hybrid designs that only use PMEM, compared with hybrid designs that use both DRAM and PMEM, preventing the achievement of practicality requirement 3 from Section 4.1. Their technique's automated conversion method also does not account for the pointer differences required to reference objects that may be relocated in virtual memory

after a crash, as is possible in the PMEM programming model, precluding requirements 2 as well.

### 4.1.3 Conversion of Lock-Free Algorithms with Non-repairing, Non-Blocking Writes

Although volatile locks can be added to Herlihy et al.'s algorithm to allow the detection of incomplete operations using RECIPE, this would require that during recovery these locks be reinitialized and reassociated with the nodes, making recovery time dependent on the size of the structure. Instead, failure detection can be done using a method similar to that proposed by Aguilera et al. using an epoch number associated with an object whose consistency is being verified [3]. This method of failure detection is implemented by Golab and Ramaraju to construct recoverable mutexes [36], and a form of it can be seen implemented in BzTree [5] just for the detection of interrupted node resizes. Using an epoch count to detect conditions is also used in other areas, including epoch-based memory reclamation [33].

Each failure-free period is tracked using a monotonically increasing PMEM-resident variable called the `epochID`. During creation or confirmation of consistency of a node, the `epochID` of the node is set to match the current epoch. After a failure, the `epochID` of a node will be out of date. Any thread that comes across it will know that there are no other threads responsible for fixing any inconsistencies with that node, which would be the case if the `epochID` did match the current epoch. This replaces the method used by RECIPE to detect if a node's inconsistencies will be repaired, which they do by checking for a held lock [47]. When traversing the structure, an outdated `epochID` is first updated by a thread trying to claim it so that it can restore consistency, preventing multiple threads from trying to repair the same node.

With the ability to discern incomplete non-blocking writes from failed ones using only PMEM-resident variables, it is now possible to follow the rest of the third RECIPE conversion method to achieve recoverability by implementing recovery methods for these inconsistencies. Helper mechanisms are added to repair inconsistencies, shown in Functions 10, 11, and 12, which are described in Section 4.4. While checking the `epochID` during traversals does not impact wait-freedom, if it is outdated due to a crash and recovery is started, the preservation of wait-freedom depends on the helper mechanisms.

### 4.1.4 Logging for Recoverable Memory Allocation

Logging for UPSkipList's memory allocations requires only that modifications that will leave memory unreachable if interrupted be logged, as all other issues can be detected during traversal and recovered as in Section 4.1.3. The thread logging its allocation only stores the latest log as well, overwriting previous ones and limiting the amount of space required. This does not add much overhead due to a skip list insertion already requiring a flush for every level that is modified in its lock-free algorithm, and due to the write latency of real PMEM modules nearly matching that of DRAM [46].

The logging method used for UPSkipList is based on the assumption that the identity of a thread performing operations does not change during an epoch. Prior to allocating memory, a thread persists a log into a thread-specific area that indicates during which epoch the allocation will be persisted, shown in the `Insert` operation in Function 4, line 38 using `LogChangeAttempt` detailed in Function 3. For all functions in this chapter, it is assumed that the unique *threadID* of the thread calling the function, the *NUMAnode* the thread is running on, and the current failure-free *epochID* are available. The persistence of a log prior to modification is similar to how a descriptor is used in BzTree [5] to indicate which links will be need to be changed to allocate memory, however in UPSkipList it is done only for a single link.

Due to the log being thread-specific, the log reading and writing cannot be blocked by other threads, leaving no impact on the deadlock-freedom of insertions, recoveries during traversals, and removes. The traversal that occurs on lines 17 to 22 occurs without taking any locks, and so like traversals in Herlihy et al.'s algorithm, cannot be blocked. In the event that an allocated node is not reachable after a crash, the deletion of that node must maintain deadlock-freedom to ensure UPSkipList's logging is deadlock-free. This is covered in Section 4.3.3.

Prior to the next modification log being written, this log is checked to see when the last thread with this identity was attempting its operation (line 14). If the epoch of the log is the same as the current one, the thread can be sure that the previous operation was a success, since the thread must have finished the operation before it could have moved on to the current operation, and the thread continues with overwriting the previous log with the log for its next operation so that the next operation can be performed. If the epoch was different, then the thread has to check whether the interrupted thread in the previous epoch had managed to succeed in its operation. This can be seen in lines 17-22, where it navigates the abstract set from the pointer that must precede this node (*bottommostPredecessorNode*) until the expected location of the key is found. If it is not found, then it cleans up any unreachable memory that has not been reallocated (line 24)

**Function 3:** LogChangeAttempt(Block *allocatedBlock*, Node *bottommostPredecessorNode*, Key *key*)

**Data:** Shared log structure logs
**Result:** In-place log creation, with incomplete allocations from past failure-free epochs cleaned up

```
13 log := logs[threadID]
14 if log is initialized and log.epochID ≠ epochID then
       /* check to see if the allocation and insertion by this thread in
          the past epoch succeeded                                      */
15     reachable := False
16     currentNode := log.bottommostPredecessorNode
17     while currentNode.keys[0] < log.key do
           /* navigate bottom-most level of the skip list from this
              pointer onwards until expected location of this key is found
              */
18         currentNode := currentNode.next[0].ToPointer()
19         if currentNode.keys[0] = log.key and currentNode = log.allocatedBlock
            then
20             reachable := True
21         end
22     end
23     if not reachable then
24         DeleteLinkedObject(log.allocatedBlock)
25     end
26 end
27 logs[threadID] := NewLogEntry(allocatedBlock, bottommostPredecessorNode, key,
     epochID)
28 Persist(logs[threadID])
```

and continues with its execution.

Using this method, after a whole-system-failure of $k$ threads, $O(k)$ recovery steps will have to be taken for memory to regain its integrity, and space is only required for $k$ logs. This way, reclamation of unreachable memory due to a crash is deferred out of recovery time and into the run time of the next operation. Since it is possible for a log to be less than a cache line in length, allocations are able to achieve recoverability with the use of a single additional cache line flush (line 28).

### 4.1.5 Recovery Time

Using the methods of this section, recovery time is reduced to the time necessary for the program to reconnect to all of its pools, after which it can immediately resume servicing requests, with inconsistencies repaired as identified during runtime. In this way, recovery is not dependent on the size of the data structure.

## 4.2 Data Structure

UPSkipList's node structure is similar to that of Herlihy et al.'s lock-free skip list, with changes made to allow recoverability, hold multiple keys per node, and to allow recoverable node splits. As previously mentioned in Section 4.1.3, an `epochID` is added to allow detection of incomplete operations. To support multiple keys per node, the key field is expanded into an array of keys stored contiguously, and the same is done with the values field. A counter is added for the number of splits that the node has undergone, which is used by the lock-free reads to confirm that the values and keys they read match. More details on this process are provided in Section 4.4. Additionally, a reader-writer lock is used to prevent updates to values from occuring while a node is being split. Details on how this is made recoverable are provided in Section 4.5. Updates that do not require a node split are concurrent with each other and reads, and building of the towers is still performed without acquiring locks, limiting the time that a node spends write-locked to the time required to transfer keys to a new, subsequent node during a node split. The remaining fields are the same as in the lock-free skip list algorithm, though since marking of deleted nodes is not used in UPSkipList, the pointers in the array of next node pointers are not markable. Like Herlihy et al.'s algorithm, head and tail sentinel nodes are used as starting and ending points for traversals.

For simplicity of development, each node and memory block is the same size, even though most nodes will use a small fraction of the available space to store pointers. The node size and memory allocator (Section 4.3.3) can be modified to reduce wasted space by allowing variable size allocations.

## 4.3 Memory Management

### 4.3.1 Persistent Pointers and NUMA Awareness

In the PMEM programming model, the physical representations of data stored in PMEM as files are memory mapped into the program's address space [58]. Due to modern requirements like address space layout randomization, and changes in a program's memory layout over versions, the data cannot be expected to be mapped to the same base address each time the data structure is used by a process, so pointers cannot be stored absolutely [15] if the practicality requirement 2 is to be satisfied.

Existing solutions to this problem include storing pointers as offsets relative to the base address of the pool, fat pointers using an additional word per pointer to store the ID of the pool in which the offset is located [23], and the Region-ID in Value (RIV) method [15]. Offsets do not allow the use of multiple pools memory mapped in at different base addresses as is the case with multiple NUMA nodes. Fat pointers, while indicating which pool of memory an offset is in, use multiple words per pointer, increasing bandwidth usage and reducing cache efficiency due to fewer pointers fitting in a single cache line. The RIV method, where the offset portion of a pointer is limited to the least significant bits, and the most significant bits are used to indicate which segment of memory the offset is relative to, is able to account for the limitations of offsets while not increasing the size of a pointer and reducing performance. The trade-off is being limited to a fraction of the number of pools; given that the 16 most-significant bits of addresses are unused on 64-bit x86 systems and the need for just one pool per NUMA node, being able to address up to $2^{16}$ pools is sufficient for UPSkipList.

The RIV method is adapted here to allow both dynamic memory allocation of segments from a shared pool of memory and to use multiple pools of memory across multiple NUMA nodes. This allows a structure to be aware that it is scaling across multiple NUMA nodes, which is not possible when NUMA support is achieved by striping a single pool across multiple nodes. The benefit of this method is compared with striping in Chapter 5.

Multi-pool NUMA awareness is done using a two-stage lookup procedure, where the top $n$ bits of a pointer are used to denote the NUMA node/pool that the memory segment

is in, the middle $m$ bits are used to denote the memory segment whose base the offset is relative to, and the bottom $64-(m+n)$ bits are the offset of the object within the memory segment. For UPSkipList, the top 16 bits that are already unused have been repurposed to represent the NUMA node, though all 16 bits are only used due to having no other purpose for them in this algorithm. If bits need to be reserved for pointer marking and tagging, the number of bits used for the NUMA node could easily be reduced to 8 or 4 bits. Dynamic memory allocation of segments can be done with any method, and need not return a pointer in RIV form. When using a single pool, whether on a single node or striped, the NUMA node lookup procedure is omitted, while lookup using segments and offsets within the single NUMA node is retained. As lookups are composed of reads and pointer arithmetic, they cannot be blocked and do not affect progress conditions.

An example of the extended RIV lookup is shown in Figure 4.3. Once the ID of the pool is determined, the next set of bits is used to determine which memory segment, or "chunk" of the pool the offset is relative to, and its absolute base address is determined. To obtain the virtual memory address where the object to be accessed is stored, the remaining least-significant bits are added to that absolute address.

### 4.3.2 Coarse-grained memory allocation

UPSkipList supports two modes of operation:

1. All memory pools are preallocated to UPSkipList. Internally, UPSkipList will prepare them by dividing each chunk into memory blocks and linking them together as a free list prior to operation.

2. Memory pools are shared with other persistent data structures used by the program and the space used by UPSkipList within the pool grows as required.

For mode 2, allocation is done at run-time as additional memory is required, allowing immediate operation, and trading some performance on inserting new nodes for the ability to grow and shrink the data structure size within the PMEM pool. For UPSkipList's implementation, `libpmemobj` is used to allocate MiB-scale chunks as `libpmemobj` objects from a pool of memory that can be shared among multiple structures in a single program, however any persistent heap allocator can be used, regardless of how their pointers are represented. Alternatively, a pool could be used that is only as big as the space required, which can be grown when more space is required. This solution however has reduced performance, as modifying the memory-mapped file requires system calls and the overhead those entail.

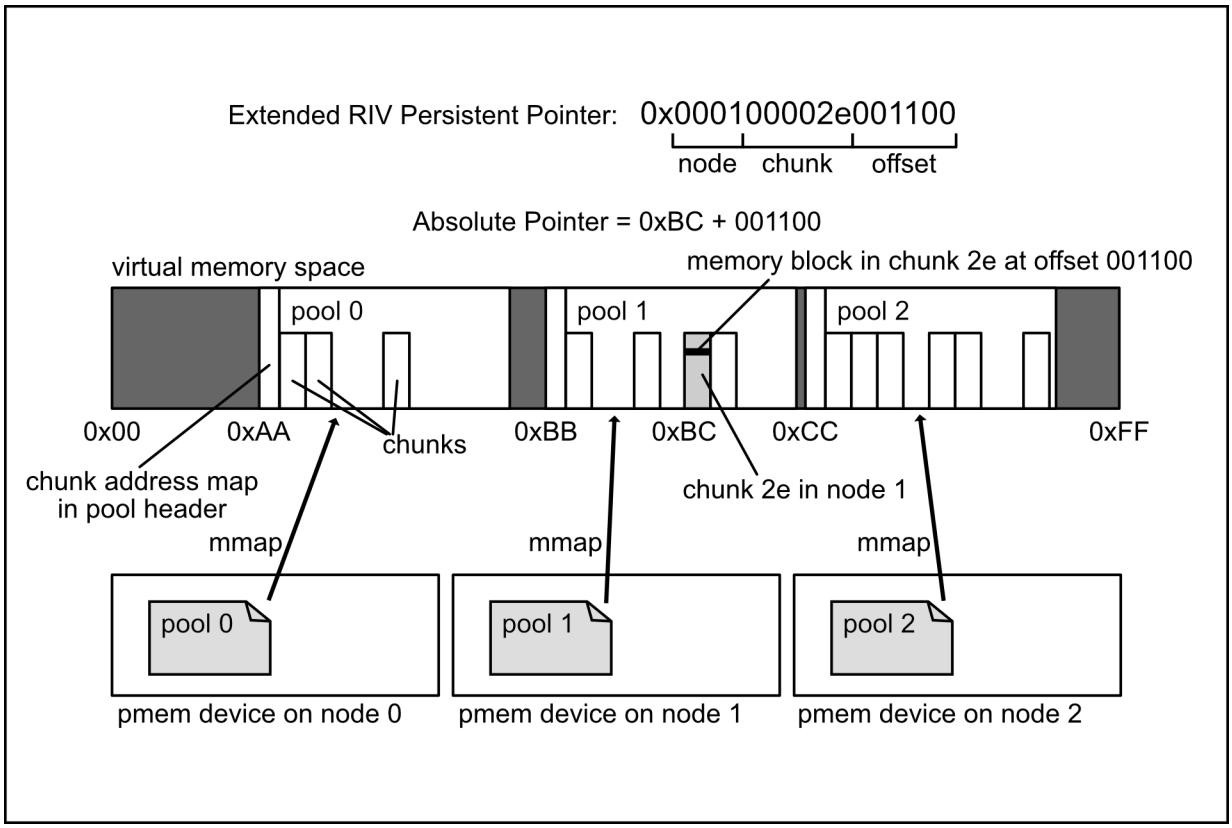


Figure 4.3: Example of extended RIV persistent pointer, and lookup process. Objects stored in chunks can be nodes or memory blocks.

Once allocated to the program by `libpmemobj`, any data stored within the segments is the data structure's responsibility. The resulting fat pointers from the `libpmemobj` shared allocation for each chunk are stored in an array in persistent memory. The absolute addresses of the chunks within the current program's address space, which do not change during a run, are cached in an array in DRAM to allow quick lookups. During recovery, the cache can be rebuilt as pointers are dereferenced, deferring it out of recovery and into when it is actually needed. This way, recoverable coarse-grained allocation is achieved within a NUMA node shared among multiple structures. This method can be extended to allow relocation of chunks within a memory pool, by updating the absolute address cache when safe.

### 4.3.3 Fine-Grained Memory Management

Memory within segments is managed manually by UPSkipList so that it can be referenced using the RIV method, and to avoid the overhead from the general memory allocation method, which is more expensive on PMEM than on DRAM, in favour of a data-structure-specific one. It is divided into memory blocks that are linked together using a free list, with each free block pointing to the next free block in the list. For recovery purposes, a memory block also contains the failure-free *epochID* in which it was created. The allocator for UPSkipList is shown in Function 4. When the pool of free blocks gets sufficiently low, as on line 34, a new chunk is requested from the coarse-grained allocator and allocated (line 35). UPSkipList relies on `libpmemobj` memory allocations to be at least deadlock-free to prevent them from impacting its own deadlock-freedom in insertion operations. Upon obtaining a new chunk, the requesting thread initializes it as a block allocator and adds all the created memory blocks to multiple lock-free free lists, before linking it into the allocator. These are all connected using the RIV method, meaning that each block in a chunk on a NUMA node can point to blocks on chunks in different NUMA nodes, which is useful when deallocating skip list nodes, for example after a failed insertion.

To reduce contention, threads on the same NUMA node will obtain new nodes to initialize from separate free lists, known as arenas, for that NUMA node, with the arena of each thread determined by the remainder after dividing its ID by the total number of free lists (line 29).

Using UPSkipList's logging method, before a node is removed from the free list, a log is made by the thread indicating where this node will be linked in on success, as explained in Section 4.1.4. In the event that a block logged from a previous epoch is found to be unreachable, the `DeleteLinkedObj` function from Function 5, called in `LogChangeAttempt`

**Function 4:** MakeLinkedObject(Node *bottommostPredecessorNode*, Array<Key> *keys*, Array<Value> *values*, int *newNodeHeight*)

**Data:** Array of headBlocks of the free lists for this node, *numberOfArenas* arbitrarily chosen for the allocator

**Output:** Pointer to newNode constructed by `MakeLinkedObject`

```
29 arenaNo := threadID % numberOfArenas
30 allocated := False
31 while not allocated do
32     newBlock := headBlocks[arenaNo]
33     nextBlock := newBlock.next
34     if nextBlock = null then
35         AllocateNewChunk(newBlock)
           /* Obtain a new chunk of memory blocks from the allocator,
              convert it into a linked list of memory blocks, then link
              its head in as the next block of newBlock                  */
36         continue
37     end
38     LogChangeAttempt(newBlock, bottommostPredecessorNode, keys[0])
       /* crashes after this point will not leak memory because the log
          will be checked and the unreachable memory will be found in a
          future allocation                                              */
39     allocated := CAS(headBlocks[arenaNo], newBlock, nextBlock)
40 end
41 Persist(headBlocks[arenaNo])
42 newNode := newBlock.InitializeAsNode(keys, values, newNodeHeight, epochID)
43 Persist(newNode)
44 return newNode
```

**Function 5:** DeleteLinkedObject(Object object)

**Data:** Array of tailBlocks of the free lists for *currentNUMANode*, *numberOfArenas* arbitrarily chosen for the allocator

**Output:** Return of object to the allocator if not already done

```
45 arenaNo := threadID % numberOfArenas
46 if object is a node then
47 |   object.ConvertToMemoryBlock()
   |   /* de-initialize the node to be deleted by zeroing it out and
   |      preparing it to be linked back into the free list          */
48 |   Persist(object)
49 else
50 |   if object = tailBlocks [arenaNo] then return
51 |   if object.next ≠ null then return
   |   /* another thread must have observed it as the tail block for it
   |      to now point to another block, so it was deleted successfully
   |      */
52 end
53 LinkInTail(arenaNo, object)
```

**Function 6:** LinkInTail(int *arenaNo*, Memory Block *newTail*)

**Data:** Array of tailBlocks of the free lists for *currentNUMANode*
**Result:** Linking of *newTail* in as the new tail of tailBlocks

```
54 while True do
55 |  currentTail := tailBlocks[arenaNo]
56 |  tailChanged := CAS(currentTail.next, null, newTail)
57 |  if tailChanged then break
58 |  if currentTail.epochID ≠ epochID then
   |  |  /* tailBlocks[arenaNo] does not point to the tail due to a
   |  |     failure; let's help it along                              */
59 |  |  nextTail := currentTail.next
60 |  |  if CAS(tailBlocks[arenaNo], currentTail, nextTail) then
61 |  |  |  Persist(tailBlocks[arenaNo])
62 |  |  end
63 |  end
64 end
65 Persist(currentTail.next)
66 CAS(tailBlocks[arenaNo], currentTail, newTail)
67 Persist(tailBlocks[arenaNo])
```

from Function 3 line 24, is called again. `DeleteLinkedObj` is idempotent, allowing recovery from a failed recovery. Composed of multiple steps that cannot be blocked, and with any failed `CAS` being attempted until success, `DeleteLinkedObj` also achieves lock-freedom, ensuring that it does not prevent any functions calling it from being lock-free themselves. If the node is not reachable and has already been deallocated but not added to the free list, which is checked on lines 50 and 51 of Function 5, the thread will continue recovery from the last step performed until it succeeds.

Deallocation, reallocation, and failure prior to clearing the log entry is accounted for by using additional metadata in the log entry. Prior to deallocating a block during recovery, additional information in the log is used to verify that it was not successfully deallocated and reallocated by a different thread. This prevents the incorrect behaviour of a thread deallocating another thread's block if a failure and reallocation has occurred. If a failure occurs during the provisioning of a new chunk, the thread will see when it attempts its next operation that the chunk being built was unsuccessfully linked in, clean it up, and return it to the coarse-grained allocator.

### 4.3.4 Memory Block Structure

Individual memory blocks within a chunk are very simply structured. For the purposes of this thesis, they are all the same size and large enough to contain a single node with the maximum number of levels. They contain a persistent pointer to the next block in the free list, and an identifier for the block itself. They also contain the epoch in which they were generated, so that an interrupted memory allocation/deallocation can be recovered.

## 4.4 Skip List Traversal

Traversals in UPSkipList, shown in Function 7, have been modified from Herlihy's algorithm to add inconsistency detection and recoverability as explained in Sections 4.1.3 and 4.1.4. Modifications have also been made to support multiple keys in a single node and to facilitate the checking of correctness of a search by tracking the number of splits a node has undergone. Internal keys are stored unordered, save for the fact that all keys are greater than the first key. To speed up searches when skip list nodes contain multiple keys, only the first key in the node is used for finding the node containing the desired key using the skip list traversal algorithm, shown on line 80 of `Traverse` in Function 7. This works because all internal keys in a node are larger than the first key in the node, and smaller

**Function 7:** Traverse(Key_t *key*, Array<Node> predNodes, Array<Node> succNodes)

**Output:** in-place population of predNodes and succNodes, *splitCount* of current node, keyIndex in node if *key* exists, *True*/*False* whether *key* was found, levelFound at which *key* was found

```
68 recoveriesDone := 0, levelFound := notFound, keyIndex := −1
69 while True do
70     predecessorNode := root.head
71     for level from topLevel down to 0 do
72         currentNode := predecessorNode.next[level]
73         successorNode := currentNode.next[level]
74         while True do
75             if CheckForRecovery(level, currentNode, predNodes, succNodes,
                recoveriesDone) then
76                 recoveriesDone += 1
77                 continue outer while loop at 69
78             end
79             currentSplitCount := currentNode.splitCount
80             if currentNode.keys[0] ≤ key then
81                 splitCount := currentSplitCount
82                 predecessorNode := currentNode; currentNode := successorNode
83                 if currentNode.keys[0] = key then
84                     keyIndex := 0, levelFound := level
85                     break
86                 end
87             else break
88         end
89         if level = 0 then
90             keyIndex := ScanInternalKeys(predecessorNode, key)
91             if keyIndex ≠ −1 then levelFound = 0
92         end
93         predNodes[level] = predecessorNode, succNodes[level] = currentNode
94         if keyIndex ≠ −1 then return ⟨ splitCount, keyIndex, True, levelFound ⟩
95     end
96     return ⟨ splitCount, keyIndex, False, levelFound ⟩
97 end
```

**Function 8:** ScanInternalKeys(Node currentNode, Key *key*)

```
   Output: index of key in currentNode.keys if key exists
98 for keyIndex from 1 up to length(currentNode.keys) do
   |  /* skip first element; it was checked in Traverse()          */
99 |  if currentNode.keys[keyIndex] = key then
100|  |  return keyIndex
101|  end
102|  return −1
103 end
```

than the first key in the next node, allowing internal keys to effectively be treated as an additional level at the bottom of the skip list. The scan of the internal keys can be seen on line 90, where the bottom-most keys are only scanned once the node that might contain the desired key has been reached on the bottom-most level. If the key is found, the index it was found at is saved so that the operation calling `Traverse` does not need to rescan the keys itself. To further improve performance, the array of keys is positioned such that the first key falls into the same cache line as additional metadata that has to be read anyway during a traversal. The scanning cannot be blocked so it does not affect the lock-freedom of Herlihy et al.'s algorithm.

The scanning process of internal keys is shown in Function 8. Storing internal keys unordered saves on overhead during insertion, where if a key is found to not exist it can be inserted using a single `CAS` operation to claim an empty slot in the node. For node sizes on the order of hundreds of keys, the overhead of having to scan all keys was found to be negligible compared to the time to find the node itself being magnitudes greater, due to hardware fetching the additional cache lines when a sequential scan is detected.

For the operation calling `Traverse` to decide if the values it reads are correct, `Traverse` records the number of splits that the current node has undergone prior to reading its first key or internal keys. This way, it can compare the number of splits prior to `Traverse` reading the keys in the node with the number of splits after it has read the keys in the node, as shown in `Search` in Function 9 on line 110. If the value does not match, then a split had occurred, the value is unreliable, and the traversal needs to be retried. Whether the node is write-locked is also checked, since a value read at the time of returning that is write-locked is also unreliable.

In the original algorithm, traversals make modifications to the skip list when they come across any nodes marked for removal by snipping their links out of the level at which they

**Function 9:** Search(Key_t *key*)

```
    Output: value corresponding to key if it exists, notFound if it does not
104 while True do
105 |   ⟨ splitCount, keyIndex, keyExists, levelFound ⟩ := Traverse(key, predNodes,
    |    succNodes)
106 |   if keyExists then
107 |   |   node := predNodes[levelFound]
108 |   |   if node.splitLock is write-locked then continue
109 |   |   value := node.values[keyIndex]
110 |   |   if node.splitCount ≠ splitCount then continue
111 |   |   return value
112 |   end
113 |   return notFound
114 end
```

have been found, one by one. To make snipping recoverable, the modification needs to be persisted, with a flush happening immediately after the next pointer of the previous node is replaced with the next one. Due to removals in UPSkipList simply tombstoning the value for the key being removed in larger, multi-key nodes, described in Section 4.6, this portion of Herlihy et al.'s algorithm has been omitted. Deleting nodes that are full of tombstones would be beneficial and is a potential future improvement to UPSkipList.

### 4.4.1 Recovery

As mentioned in Section 4.1.3 and shown in Function 10, during runtime operation, each node's `epochID` is checked to determine whether any inconsistencies have to be repaired by the observing thread (line 116). If the `epochID` matches that of the current run, then the thread knows that either this node is consistent, or it will be made consistent by another thread that's still working on it, and does not bother checking for inconsistencies. There is minimal overhead to checking this as it is contained in the same cache line as other important metadata that need to be read regardless during traversal.

If the `epochID` does not match, the thread needs to clear any outdated metadata, shown on line 122 where the reader count from the previous epoch is reset, and then claim the right to check/repair that node by performing a CAS operation from the old `epochID` to the new `epochID` (line 123). The metadata has to be reset prior to updating the epoch

**Function 10:** CheckForRecovery(int *level*, Node currentNode, Array<Node> predNodes, Array<Node> succNodes, *recoveriesDone*)

```
    Output: True if a recovery has been done, False otherwise
115 nodeEpoch := currentNode.epochID, recoveryNeeded := False
116 if nodeEpoch ≠ epochID then
117     if currentNode.splitLock is read- or write-locked then
118         oldReaderCount := currentNode.splitLock.GetReaderCount()
119         recoveryNeeded := True
120     end
121     if recoveriesDone = 0 or recoveryNeeded then
122         splitLock.DrainReaders(oldReaderCount)
123         if not CAS(currentNode.epochID, nodeEpoch, root.epochID) then
124             return False
125         end
126         CheckForNodeSplitRecovery()
127         CheckForInsertRecovery(level, currentNode, predNodes, succNodes)
128         return True
129     end
130 end
131 return False
```

so that the outdated state does not become visible to concurrent operations that assume it is current. By having a single thread claim the node by updating the epoch, multiple threads will not end up trying to repair the same node, and other threads can be sure that any node they come across that has a current `epochID` has a thread that will fix any inconsistencies in that node, letting it be treated the same as concurrent operations prior to the failure. Threads that fail to claim a node for recovery continue with their original operations, and are not blocked.

Once an old node has been claimed, the thread will now check to see whether it is inconsistent (lines 126 and 127), and if so, it will repair it (Function 11 lines 133–147, Function 12 line 151). Both these functions recover incomplete operations by completing the missing steps. `LinkHigherLevels` on line 151 in Function 12 is used in Herlihy et al.'s original algorithm, and therefore is lock-free. `CheckForNodeSplitRecovery` in Function 11 cannot be blocked either, proceeding to complete a split if the node in question is found to be write-locked and then unlocking it. These functions releasing node split locks from a previous failure-free epoch are crucial in maintaining the deadlock-freedom of UPSkipList

**Function 11:** CheckForNodeSplitRecovery(Node currentNode)

**Result:** currentNode in consistent state of either a failed or successful node split

```
132 if currentNode.splitLock is write-locked then
133     succNode := currentNode.next[0]
134     for keyIndex from 0 up to length(currentNode.keys) do
135         if currentNode.keys[keyIndex] = null then
136             currentNode.values[keyIndex] := tombstone
137         else
138             for succKeyIndex from 0 up to length(succNode.keys) do
139                 if currentNode.keys[keyIndex] = succNode.keys[succKeyIndex] then
140                     currentNode.keys[keyIndex] := null
141                     currentNode.values[keyIndex] := tombstone
142                 end
143             end
144         end
145     end
146     Persist(currentNode)
147     currentNode.WriteUnlock()
148 end
```

**Function 12:** CheckForInsertRecovery(int *level*, Node currentNode, Array<Node> predNodes, Array<Node> succNodes)

**Result:** currentNode with levels linked up to its height

```
149 previousLevel := level + 1
150 if succNodes[previousLevel].key < currentNode.key then
151     LinkHigherLevels(predNodes, succNodes, currentNode, previousLevel,
        currentNode.height)
152 end
```

during node splits, as otherwise threads would be blocked forever waiting for a split to finish.

Every operation that may modify the skip list, which in this case is an insert or node split, needs to have a method for checking their completeness added, as prescribed by RECIPE [47]. As traversals are the first step in all other operations in a skip list, no attempts at recovery have to be made outside of this operation. To all other operations, this restores the expectation that any inconsistencies seen will eventually be repaired, allowing traversals to forgo additional consideration and operate as before.

#### Preventing Low Throughput After Recovery

After a failure and recovery, all nodes will now be from the previous failure-free epoch, and require updating their `epochID` and either have their consistency verified or made consistent. Performing this for every node that a thread comes across immediately after recovery can result in low post-recovery performance, in particular due to the need to flush the update before moving on. An optimization can be made that a single skip list traversal, if possible, will only attempt to repair $k$ possible inconsistencies, either sequentially or at random, where $k$ can be as low as 1. Not all inconsistencies can have their recovery deferred, however. Node splits, for example, must be repaired as soon as they are found, due to their inconsistent contents making traversals invalid unless repaired. The deferral of some recoveries is done in `CheckForRecovery` from Function 10, on line 121. This way, given enough time, the structure will recover to a consistent state, as threads can be sure that any missed, unclaimed inconsistencies will eventually be repaired at some point in the future. UPSkipList lets traversals for the purpose of searches perform one recovery of an incomplete insertion per operation, on the first unrecovered node they find. Incomplete splits are still recovered whenever found.

## 4.5 Insertion and Updates

Herlihy et al.'s insertion algorithm requires minor modification to be made recoverable using UPSkipList's logging method from Section 4.1.4 and the extension to RECIPE from Section 4.1.3. As explained in Section 4.1.4, when a thread is allocating a new node, which is a modifying operation, it checks its log to see if its previous operation had succeeded and was made visible, and if so, logs that it is doing a new allocation and continues. In the event where the previous log indicates an insertion was attempted, it is considered a "success" if the linking of the new node to the abstract set at the bottom-most level in

the expected location has succeeded. If this is the case, the current insertion continues normally, without checking if the previous insertion's node was built up to its full height, since recovery for an incomplete node is handled in `Traversal` from Function 7, described in Section 4.4. If the node is found to be unlinked, it is cleaned up and deallocated, preventing the memory from being leaked.

`Insert` is shown in Function 13. The algorithm shown here is actually an "upsert" operation, since it falls back to updating the value of a key if that key is found to already exist. To facilitate storing multiple keys in a node, several modifications have been made to Herlihy's insertion algorithm. First, if the key is found to exist, a read lock is obtained on the node and the split count is confirmed to match the count from the traversal; upon failure, the operation is reattempted, as shown on line 159. In this case, this thread's progress has been prevented by another thread progressing in a way that this thread cannot block, preventing deadlock. The read-lock prevents updates to an existing key from being performed while a node split is occurring on the node containing it, which could cause nonlinearizable behaviour. The update is performed on line 160 using `Update` from Function 14, which attempts to `CAS` the current value to the new value and return the current value until it succeeds, establishing a total order for all updates on the same key. Updates can be done concurrently on different keys in a node.

If the key does not already exist, then an attempt is made to insert the key into the node. Since the head node is just a sentinel node and does not store keys, instead of inserting the key into this node a new node is created and inserted after it using Herlihy's existing algorithm, made recoverable, as shown in `CreateHeadSuccessor` in Function 15. This maintains the lock-freedom of Herlihy's existing algorithm. Flushes are added after several steps to facilitate recovery. First, after allocating, the populated next pointers of the new node in Function 18 have to be persisted before the node can be made reachable to other threads. Since prior to the node being linked in these pointers are unreachable, and since a failure at any point prior to the new node being made reachable will be reclaimed by Function 3, the order of persistence does not matter and can be done with a single flush on line 246. Next, a flush needs to be added after the linking at Function 15 line 190 and also in Function 20 line 255, as in RECIPE. Finally, during the tower-building stage of construction, in Function 17, all next pointers of predecessor nodes have to be individually set and persisted from the bottom-most level up, which is done on lines 231 to 233. For recovery purposes, the order in which they are persisted matters, as missing lower levels in nodes that are not logically deleted are not permitted in Herlihy et al.'s algorithm.

If the node that was found is not the head node, then first an attempt is made to insert into the node. This procedure is shown in Function 16. First, the thread acquires a read lock and verifies that the split count has not changed from when it was read in Function 7,

**Function 13:** Insert(Key *key*, Value *value*)

**Output:** Existing value if updating, *null* if newly inserted

```
153 predNodes := [ ]
154 succNodes := [ ]
155 while True do
156 |  ⟨ splitCount, keyIndex, keyExists, levelFound ⟩ := Traverse(key, predNodes,
    |   succNodes)
157 |  predNode := predNodes[0]
158 |  if keyExists then
159 |  |  if not predNode.ReadLock() or predNode.splitCount ≠ splitCount then
    |  |   continue
160 |  |  oldValue := Update(keyIndex, value, predNodes[levelFound])
161 |  |  predNode.ReadUnlock()
162 |  |  return oldValue
163 |  end
164 |  if predNode = root.head then
165 |  |  if CreateHeadSuccessor(key, value, predNodes, succNodes) then
166 |  |  |  return null/* the key and value were newly inserted into a
    |  |  |     newly created node, so return null                      */
167 |  |  else
168 |  |  |  continue
169 |  |  end
170 |  else
171 |  |  status := InsertIntoExistingNode(key, value, predNodes, succNodes,
    |  |   splitCount)
172 |  |  if status = continue then continue
173 |  |  else if status = needSplit then SplitNode(key, value, predNodes,
    |  |   succNodes)
174 |  |  else return status
175 |  end
176 end
```

**Function 14:** Update(int *keyIndex*, Value_t *value*, Node predNode)

**Output:** existing value of *key*

```
177 while True do
178 |   oldValue = predNode.values[keyIndex]
179 |   if CAS(predNode.values[keyIndex], oldValue, value) then
180 |   |   Persist(predNode.values[keyIndex])
181 |   |   return oldValue
182 |   end
183 end
```

**Function 15:** CreateHeadSuccessor(Key_t *key*, Value_t *value*, Array<Node> predNodes, Array<Node> succNodes)

**Output:** linking in of newNode after the head node on all levels, *True*/*False* on whether it was successful

```
184 newNodeHeight := random integer from geometric distribution with p = 0.5
185 succNode := succNodes[0]
186 keys := [key, null, ..., null]
187 values := [value, tombstone, ..., tombstone]
188 newNode := MakeLinkedObject(head, keys, values, newNodeHeight, epochID)
189 PopulateNextPointers(succNodes, newNode, newNodeHeight)
190 linkedIn := CAS(head.next[0], succNode, newNode)
191 if linkedIn then
192 |   Persist (head.next[0])
193 else
194 |   DeleteLinkedObject(newNode)
195 |   return False
196 end
197 LinkHigherLevels(predNodes, succNodes, newNode, 1, newNodeHeight)
198 return True
```

**Function 16:** InsertIntoExistingNode(Key *key*, Value *value*, Array<Node> predNodes, Array<Node> succNodes, int *splitCount*)

**Output:** *status* $\in$ {*continue*, *needSplit*, *oldValue*} indicating whether the `Insert` operation calling this function has to restart, split the node, or if *key* and *value* have been inserted, in which case *oldValue* is returned

```
199 predNode := predNodes[0]
200 if not predNode.ReadLock() then return continue
201 if predNode.splitCount ≠ splitCount then
202 |   predNode.ReadUnlock()
203 |   return continue
204 end
205 for keyIndex from 0 up to length(predNode.keys) do
206 |   currentKey := predNode.keys[keyIndex]
207 |   if currentKey = key then
208 |   |   oldValue := Update(keyIndex, value, predNode)
209 |   |   predNode.ReadUnlock()
210 |   |   return oldValue
211 |   else if currentKey = null then
212 |   |   if CAS(currentKey, null, key) then
213 |   |   |   Persist(currentKey)
214 |   |   |   while True do
215 |   |   |   |   oldValue := predNode.values[keyIndex]
216 |   |   |   |   if CAS(predNode.values[keyIndex], oldValue, value) then
217 |   |   |   |   |   Persist(predNode.values[keyIndex])
218 |   |   |   |   |   predNode.ReadUnlock()
219 |   |   |   |   |   return oldValue
220 |   |   |   |   end
221 |   |   |   end
222 |   |   end
223 |   end
224 end
225 predNode.ReadUnlock()
226 return needSplit
```

**Function 17:** LinkHigherLevels(Array<Node> predNodes,
Array<Node> succNodes,
Node newNode, int *startingLevel*, int *newNodeHeight*)

**Result:** linking in of newNode on levels *startingLevel* up to *newNodeHeight*

```
227 for level from startingLevel up to newNodeHeight do
228     while True do
229         predecessorAtThisLevel := predNodes[level]
230         successorAtThisLevel := newNode.next[level]
231         linkedIn := CAS(predecessorAtThisLevel.next[level], successorAtThisLevel,
              newNode)
232         if linkedIn then
233             Persist(predecessorAtThisLevel.next[level])
                /* for correctness of the algorithm, it is important that
                   changes to next pointers at a level are persisted prior
                   to changes at higher levels                              */
234         else
235             succNodes := []
236             Traverse(newNode.key, predNodes, succNodes)
                /* this will repopulate the arrays with more recent
                   information, correct at the time that the links were
                   followed by Traverse()                                   */
237             PopulateLevels(succNodes, newNode, level, newNodeHeight)
238         end
239     end
240 end
```

**Function 18:** PopulateLevels(Array<Node> succNodes, Node newNode, int *startingLevel*, int *endingLevel*)

**Result:** in-place population of newNode's next pointers from *startingLevel* up to *endingLevel*

```
241 for level from startingLevel up to endingLevel do
    |  /* populate the next pointers of newNode                    */
242 |  successorAtThisLevel := succNodes[level]
243 |  newNode.next[level] := successorAtThisLevel
244 end
245 for level from startingLevel up to endingLevel do
246 |  Persist(newNode.next[level])
247 end
```

**Function 19:** PopulateNextPointers(Array<Node> succNodes, Node newNode, int *newNodeHeight*)

**Result:** in-place population of all of newNode's next pointers

```
248 PopulateLevels(succNodes, newNode, 0, newNodeHeight)
    /* the next pointers of newNode contain pointers to its successors on
       every level, from level 0 to its height                     */
```

as in `Update` from Function 14. Then, the thread scans all the keys until it finds an empty slot. It is alright to insert into the first empty slot found without scanning to see if the key exists in later filled slots in the node because the full scan was already done in `Traverse`, and because multiple threads inserting into the same node will be fighting for the same next empty slot, meaning that if the key is inserted by another thread, it will be found by this one. An insertion first claims a slot by changing its key from empty to the new key. It then updates the value. The linearization point of an insert is upon the persistence of an updated value, with the expectation that another thread reading the value forces it to be persisted by flushing it. If one thread changes the key but then another thread finds that key and updates its value prior to the first thread, then the second thread has to be treated as the inserting thread and the first thread becomes the updating thread. If while doing the scan or after failing to claim a slot the slot contains a key that matches the key to insert, this means that another thread beat this one to inserting the value, and this thread falls back to performing an update.

If it is found that no room exists, then a node split is necessary. The read lock is released, and the split is performed as shown in `SplitNode` in Function 20. Since the reader-writer lock is only to prevent updates from occurring during a node split, releasing the read lock to acquire the write lock is safe. Population of the predecessor and successor node arrays occurred without acquiring the lock. To begin the split, the write lock is obtained to prevent concurrent updates, since the node being split will be in an inconsistent state while keys are being transferred and deleted. If the write-lock cannot be obtained, that means that another thread is making progress with its insertion or update. This thread restarts the insertion operation from the beginning. In this way, deadlock-freedom is maintained. Then, a new node is created, and it is filled with all the key-value pairs where the key is larger than the median key, with the median key becoming the first key of the new node. Next, the node is linked in. To complete the split, all the key-value pairs copied over have to be erased from the original node by replacing their keys with null and replacing their values with tombstones. Once this is done, the write lock can be released, and the tower for the new node can be built. A node split ends with the `Insert` function retrying from the beginning, and inserting into the old node or the new node.

### 4.5.1 Recoverable Node Splits

Recovery from an interrupted node split requires checking to see if the successor to the node being split contains any duplicate keys. In the event that the node split had progressed to the point that the successor was inserted, then to complete the split requires finding and deleting all duplicate keys from the prior node. If the successor had not been successfully

**Function 20:** SplitNode(Key *key*, Value *value*, Array<Node> predNodes, Array<Node> succNodes)

**Result:** linking in of new node containing latter half of the keys of predecessor node

```
249 predNode := predNodes[0]
250 if not predNode.WriteLock() then return
251 newNodeKeys := second half of sorted predNode.keys followed by null keys
252 newNodeValues := corresponding values to newNodeKeys from predNode followed
     by tombstone values
253 newNode := MakeLinkedObject(predNode, keys, values, newNodeHeight,
     root.epochID)
254 PopulateNextPointers(succNodes, newNode, newNodeHeight)
255 linkedIn := CAS(predNode.next[0], newNode.next[0], newNode)
256 if linkedIn then
257 |   Persist(predNode.next[0])
258 else
259 |   DeleteLinkedObject(newNode)
260 |   predNode.WriteUnlock()
261 |   return
    |   /* if inserting the new node fails, simply return; the calling
    |      Insert function will start over again from 155 regardless    */
262 end
263 predNode.splitCount += 1
264 Persist(predNode.splitCount)
265 replace all keys in predNode that were moved to newNode with null
266 replace all values in predNode that were moved to newNode with tombstone
267 Persist(predNode)
268 predNode.WriteUnlock()
269 ⟨ splitCount, keyIndex, keyExists, levelFound ⟩ := Traverse(newNode.keys[0],
     predNodes, succNodes)
270 LinkHigherLevels(predNodes, succNodes, newNode, 1, newNodeHeight)
    /* the calling Insert function will now start over from line 155   */
271
```

inserted, then it will be cleaned up using UPSkipList's logging method described in Section 4.1.4, and the scan will find no duplicate keys to erase. A check also has to be done to ensure that keys being deleted were not only partially deleted; this is done by ensuring that any null keys have their corresponding value slots filled with tombstones. The construction of the tower of the split node is recovered using Function 12, and so does not have to be dealt with here, since it will be detected and repaired.

### 4.5.2 Recovery

After addition to the abstract set by linking the node in by `Insert` in Function 15 on line 190 and Function 20 line 255, the thread in Herlihy's algorithm proceeds to build up links level by level until the top-most level for this node is reached, using Function 17. During normal operation, any threads that come across an incomplete node can be sure that this node is being built by another thread, and can move on. In the event of an interruption, this is no longer the case, so a method is required such that traversals can detect that a node is incomplete because it is not being built up, as explained in Section 4.1.3. This is shown in Function 10. First, as mentioned in Section 4.4, nodes are checked to see whether no other thread can be responsible for them, which will be the case if they were last updated during the current epoch.

Since traversals start from the top down, if a node is first found from a previous epoch at a level that is not its top level (line 126), then it must be incomplete and needs to be built up. This is done using part of the existing `Insert` function, Function 17, which uses the existing search results in `Traverse` and links in the node to the height it ought to be at by swapping out the previous node's next pointers at each higher level to point to this one. In the event that this takes too long and things have changed, the search has to be retried (line 236 in Function 17) up until it finds this node, as would be the case with an insert.

## 4.6 Removals

Removals are done simply by replacing the removed value with a tombstone to indicate it should be treated as removed. This allows them to be performed effectively as an update. While the usage of tombstones is not the most practical, they greatly simplify the removal of single keys within nodes by effectively returning the field of the key to an uninserted state. If all keys in a node are deleted, the node has to be unlinked using Herlihy et al.'s

algorithm. Usage of a recoverable memory reclamation method to allow removals is a logical next-step for this research. With recoverable memory reclamation, UPSkipList's logging (Section 4.1.4) and runtime failure detection (Section 4.1.3) can otherwise be applied as easily for removals as it has been for insertions and updates. A log needs to be made prior to removing a node from the abstract set and returning it to the allocator, and an integrity check needs to be performed during traversal to ensure that next pointers marked during a previous failure-free epoch are part of a remove that has been linearized, and so can be snipped out safely.

The check for whether an insertion had succeeded will also need to be modified to account for whether it did succeed and the inserted node was subsequently removed. If reference counting is used for garbage collection, this can be done by counting the log as a reference to the node, preventing its deletion and reallocation until the log is cleared. In the event that an insertion has been found that is not linked in, the thread then simply has to check whether the object it had attempted to insert in a past epoch has since been removed, indicating that the insertion had completed successfully.

# Chapter 5

# Evaluation

Empirical evaluation of performance of UPSkipList was done to determine the effectiveness of the techniques from Chapter 4, and to determine the impact of the tradeoffs of the design decisions. For performance testing, both runtime and recovery-time performance was measured and compared against BzTree [5] and a `libpmemobj`-based [22] implementation of a skip list, which are both recoverable index implementations. YCSB [24] was used to generate workloads for the runtime and recovery tests.

## 5.1 Methodology

This section explains the environment in which testing was done, the description of each test, and methodology used to perform each of them.

### 5.1.1 Environment

Performance testing was done on an 80-core, 4-socket Intel Xeon Gold 6230 machine with 768 GiB of DRAM and 3072 GiB of Intel Optane DC Persistent Memory. The machine was running Ubuntu 20.04.2 LTS with 5.4.0-65-generic kernel. The program was compiled using GCC 9.3.0 with optimization level `O3`. DRAM and PMEM are divided up equally among all four NUMA nodes, with half of the PMEM on each node set up as a separate device, and the other half on all nodes combined to form a single PMEM device striped across all four nodes with a stripe size of 2 MB. The PMEM devices were formatted with XFS with DAX enabled.

### 5.1.2 Workloads

The Yahoo Cloud Serving Benchmark (YCSB) [24] was used to generate workloads with a representative fraction of writes and a realistic distributions of keys, with insert, update, and read key-value pair operations. The workloads are described in Table 5.1, with workload names and letters used interchangeably throughout this chapter. Workloads with removes were not included because in UPSkipList removes are currently implemented as an update that replaces a value with a tombstone, placing competing data structures with removals that reclaim memory at a disadvantage for throughput. During testing, each workload was memory-mapped into DRAM by the tester program, divided among threads, and played back to perform the operations. This was done to remove the overhead of workload generation from the runtime of the test. The generated workloads used 100 million key-value pairs. Threads were assigned to NUMA nodes in a round-robin manner, ensuring an equal distribution across all nodes for all multiples of four threads. All physical cores were filled out first, with remaining threads being assigned to hyperthread siblings.

| Workload | Name | Read/Update/Insert Ratio | Distribution |
|---|---|---|---|
| A | Update-Heavy | 50/50/0 | Zipfian |
| B | Read-Mostly | 95/5/0 | Zipfian |
| C | Read-Only | 100/0/0 | Zipfian |
| D | Read-Latest | 95/0/5 | Latest |

Table 5.1: Properties of YCSB workloads used for testing.

For the purposes of this test, three data structure implementations were chosen, set with the following parameters:

- UPSkipList, on both a single pool on the PMEM device striped across multiple nodes, and on multiple pools with one on each node. Experiments were run, with 256 key-value pairs per node, 32 levels, and 4 MiB chunk size for coarse-grained allocation. These values were found to have the best performance through trial and error.
- BzTree, a `PMwCAS`-based index data structure, with parameters set the same as in their original paper [5].
- A `libpmemobj` lock-based skip list converted from Herlihy's lazy skip list using PMDK's recoverable transactions, on the striped device.

BzTree was chosen as a point of comparison because it has non-blocking writes with blocking node-splits, just like UPSkipList. Its usage of `PMwCAS` removes the need to repair

inconsistencies by keeping them hidden from the program. The implementation of BzTree used is by Lersch et al., who benchmarked index data structures on real persistent memory, and found BzTree to perform the best of the PMEM-only data structures they tested [50]. Due to internal limitations of the `PMwCAS` library used by Lersch et al., this implementation of BzTree does not support more than 120 threads. The lock-based `libpmemobj` skip list was also used, as an example of what can be built using the transactional PMEM programming techniques as prescribed by the PMDK [23]. It is adapted directly from Herlihy's lazy skip list [40], so does not store multiple keys per node. It also allows comparison of the performance of `libpmemobj`'s fat pointer system with the extended-RIV system used by UPSkipList.

Throughput testing measured performance by first pre-loading the data structure and then running the workload for some time to warm up the caches and reach a steady level of performance.

Recovery tests similarly involve preloading the structure and running operations on it, with the addition of a crash at a random point, leaving any ongoing operations in an incomplete state. After this, the test program is restarted and reconnects with the data structure, recording the time it takes for the data structure to be able to respond to new requests.

## 5.2 Results and Discussion

This section contains results of the tests from the previous section, discussion of why certain data structures performed better than others, and evaluation of various design decisions.

### 5.2.1 Throughput Comparison

The results of throughput testing of the three data structures, all on the striped device, are shown in Figures 5.1 and 5.2. Each point is the average of three runs, and error bars indicate one standard deviation. These results show that our conversion technique results in a recoverable data structure that is competitive with existing work. BzTree outperforms UPSkipList at read-only workloads by on average 93% and read-latest workloads by on average 56%, as seen in Figure 5.2. The reason for this is due to BzTree having a more efficient lookup process inside nodes. In BzTree, after a node split, both nodes contain sorted keys, while keys inserted between splits are stored unsorted in an overflow region. BzTree's lookup process takes advantage of this fact, using a binary search within the

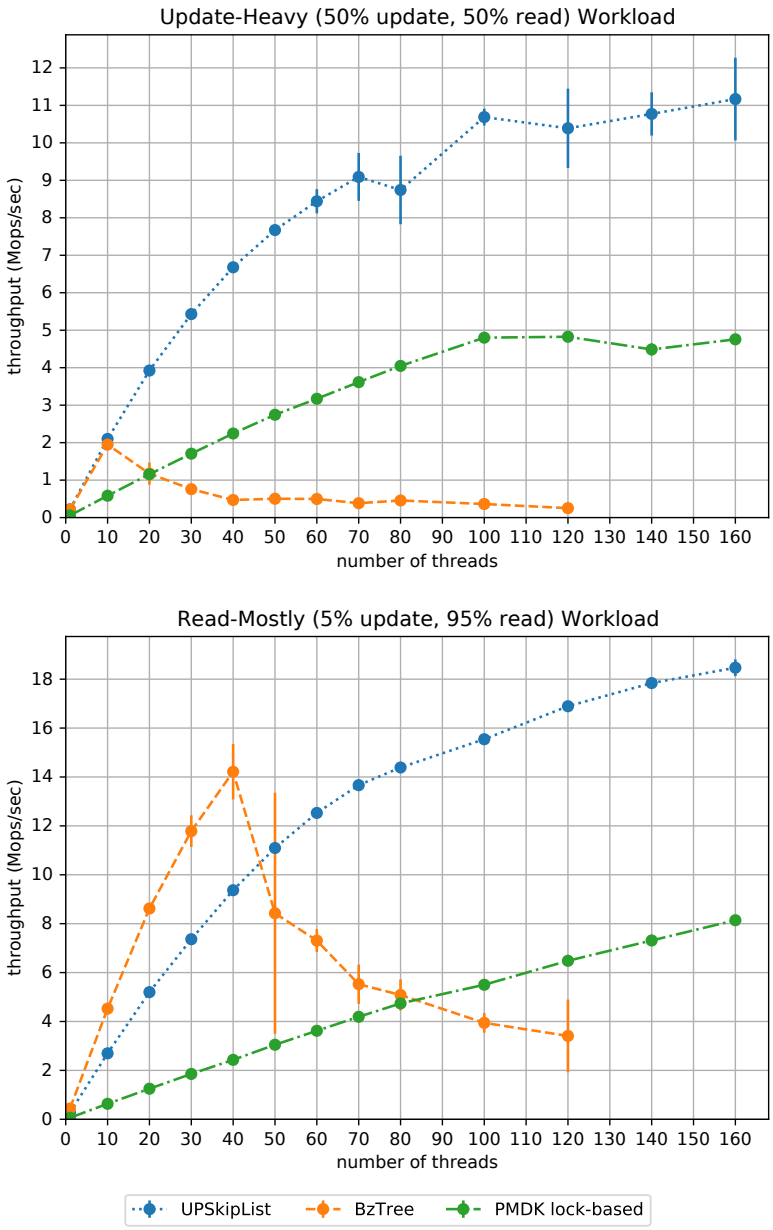


Figure 5.1: Throughput comparison using YCSB benchmark workloads A and B for UPSkipList, BzTree, and the PMDK lock-based skip list.

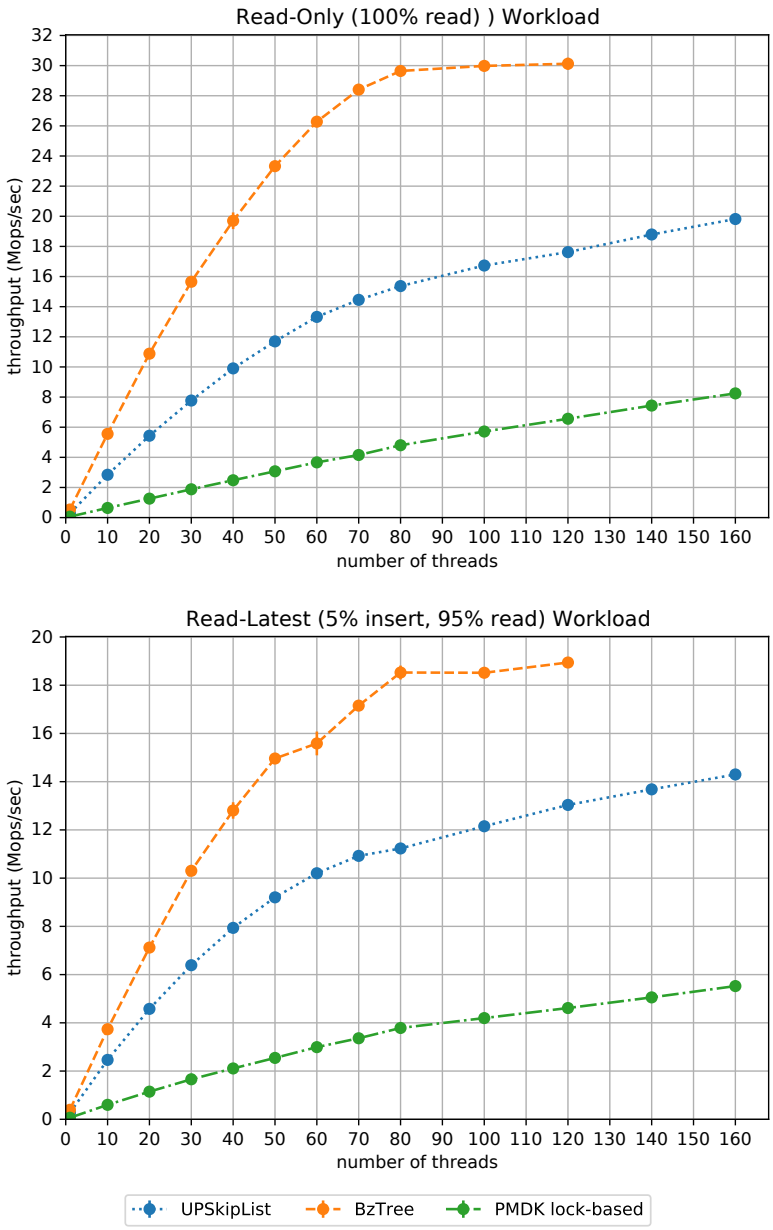


Figure 5.2: Throughput comparison using YCSB benchmark workloads C and D for UP-SkipList, BzTree, and the PMDK lock-based skip list.

sorted keys to attempt to find a key in $O(\log(n))$ time. Failing this, BzTree does a linear search on keys in the overflow region. UPSkipList currently maintains keys in an unsorted manner, so it must do a linear search to find them. The sorting optimization can be implemented in UPSkipList, due to similarity in the implementation of node splits.

In comparison, BzTree does poorly at higher update-to-read ratios, which is a result matching that of Lersch et al. [50]. This result is shown in Figure 5.1, where BzTree underperforms UPSkipList by 76% in update-heavy workloads and 3% in read-mostly workloads. Similar to their observations, BzTree scales up to a point, after which performance falls off, unlike when BzTree outperformed UPSkipList, where UPSkipList continued to scale, albeit more slowly. The level of concurrency BzTree scales up to is inversely correlated with the proportion of updates. The reason BzTree's performance falls off is due to its internal use of `PMwCAS` to perform its writes. Where UPSkipList manages to update a key using a single CAS operation, a BzTree thread needs to use `PMwCAS` to change the key value to ensure it does not interfere with a concurrent `PMwCAS` operation and can perform the update safely. Reading descriptors and helping finish their operations requires interaction with data structures internal to `PMwCAS`, which does not impact performance until reaching a certain level of contention. This is a tradeoff of using `PMwCAS` to perform non-blocking writes, which adds a potential bottleneck for some patterns of usage in exchange for simpler algorithms and correctness. Adapting an existing, mature, non-recoverable algorithm using RECIPE also provides simpler implementation and correctness, and the extension to detect inconsistencies in algorithms with non-blocking, non-repairing writes reduces the need to use external libraries that may add bottlenecks like `PMwCAS`.

Compared to the lock-based, `libpmemobj`-based skip list, UPSkipList outperforms it in all scenarios, more than doubling its throughput, seen in Figures 5.1 and 5.2. This suggests that when better performance is desired, adapting a lock-free algorithm using the extension to RECIPE from Section 4.1.3 is more effective than using transactions to implement a lock-based algorithm.

More interestingly, it can be seen that the `libpmemobj` skip list outperforms BzTree at higher levels of concurrency at both the update-heavy and read-mostly workloads, as shown in Figure 5.1. This is for the same reason that UPSkipList outperforms BzTree, with BzTree's performance falling due to its use of `PMwCAS`, which becomes a bottleneck at higher concurrency. While `libpmemobj` skip list's use of locks causes it to plateau in performance due to contention above 100 threads in the update-heavy workload, the impact due to contention is much lower than `PMwCAS` and suggests that when good performance is required across a range of workloads, lock-based `libpmemobj` may indeed be a better choice than lock-free `PMwCAS`.

### 5.2.2 `libpmemobj` vs RIV Pointers

An additional comparison can be done using the throughput results of the `libpmemobj` skip list to determine the overhead of its pointer system. As mentioned before, `libpmemobj` uses pointers that are two words wide, with the first word containing an identifier for the pool and the second word containing the offset within that pool. Dereferencing these pointers requires a lookup of the pool's base address, just like for RIV pointers in UPSkipList, but more importantly their double width reduces the number of nodes that can fit in cache by roughly a factor of 2.

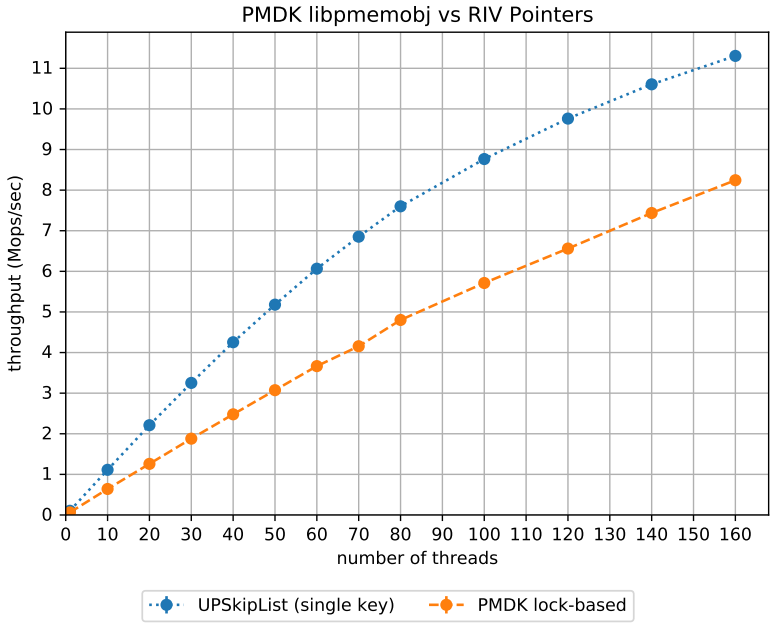


Figure 5.3: Comparison of read-only throughput of UPSkipList with a single key per node, using RIV pointers, with the lock-based skip list, using PMDK's `libpmemobj` fat pointers.

Figure 5.3 shows the throughput comparison of the `libpmemobj` skip list and UPSkipList in a read-only workload. The read-only workload does not modify the skip list and only involves following pointers, allowing the comparison of different pointer representation methods implementing the same structure. To match the structure of the `libpmemobj` skip list, UPSkipList is only storing a single key per node in this experiment. It can be seen that the use of fat pointers by `libpmemobj` compared to RIV pointers used by UPSkipList is detrimental to performance, achieving at most around 70% of the throughput due to the

reduced cache efficiency storing fewer pointers to the next nodes in the tower per cache line.

### 5.2.3 NUMA-aware vs Striped Performance

When running on the striped device, only a single pool of memory is revealed to UPSkipList. While reads and writes are distributed across multiple NUMA nodes simply by using the striped device, in this mode UPSkipList is not able to know anything about which NUMA node any of its skip list nodes are stored on. Running with multiple pools, with one on each node, UPSkipList is aware of which NUMA node each of its reads and writes are interacting with, and manually ensures they are spread out to reduce load on any one node. The purpose of this comparison is to examine the impact of NUMA awareness on performance, as NUMA awareness allows the implementation of algorithms that can make more accesses local than the $1/n$ limit that is possible on a device striped across $n$ NUMA nodes. This lower level of abstraction can allow better performance with a smarter algorithm that strategically places data on the NUMA nodes that are accessing them the most, and avoids accessing distant data if possible. Algorithms of this sort using skip lists exist [27, 59], and are a logical next step to improve performance, given that NUMA-awareness is not too expensive.

Figure 5.4 compares the throughput of UPSkipList running on the striped device with UPSkipList running on multiple pools. It can be seen that the performance impact is indeed very small across all workloads, averaging 5.6% as shown in Table 5.2, with the greatest variation falling within the margins of error for both implementations. This suggests that while striping is a great way to make non-NUMA-aware structures benefit from running on NUMA machines, explicit awareness via the extended RIV pointer method is a viable technique to allow the implementation of algorithms in persistent memory that take advantage of NUMA locality to improve performance and scalability.

| Workload | A | B | C | D | Average |
|---|---|---|---|---|---|
| Throughput reduction | 5.1% | 5.6% | 5.9% | 6.0% | 5.6% |

Table 5.2: Performance impact of running UPSkipList on multiple pools with NUMA awareness compared to running on a single, striped pool.

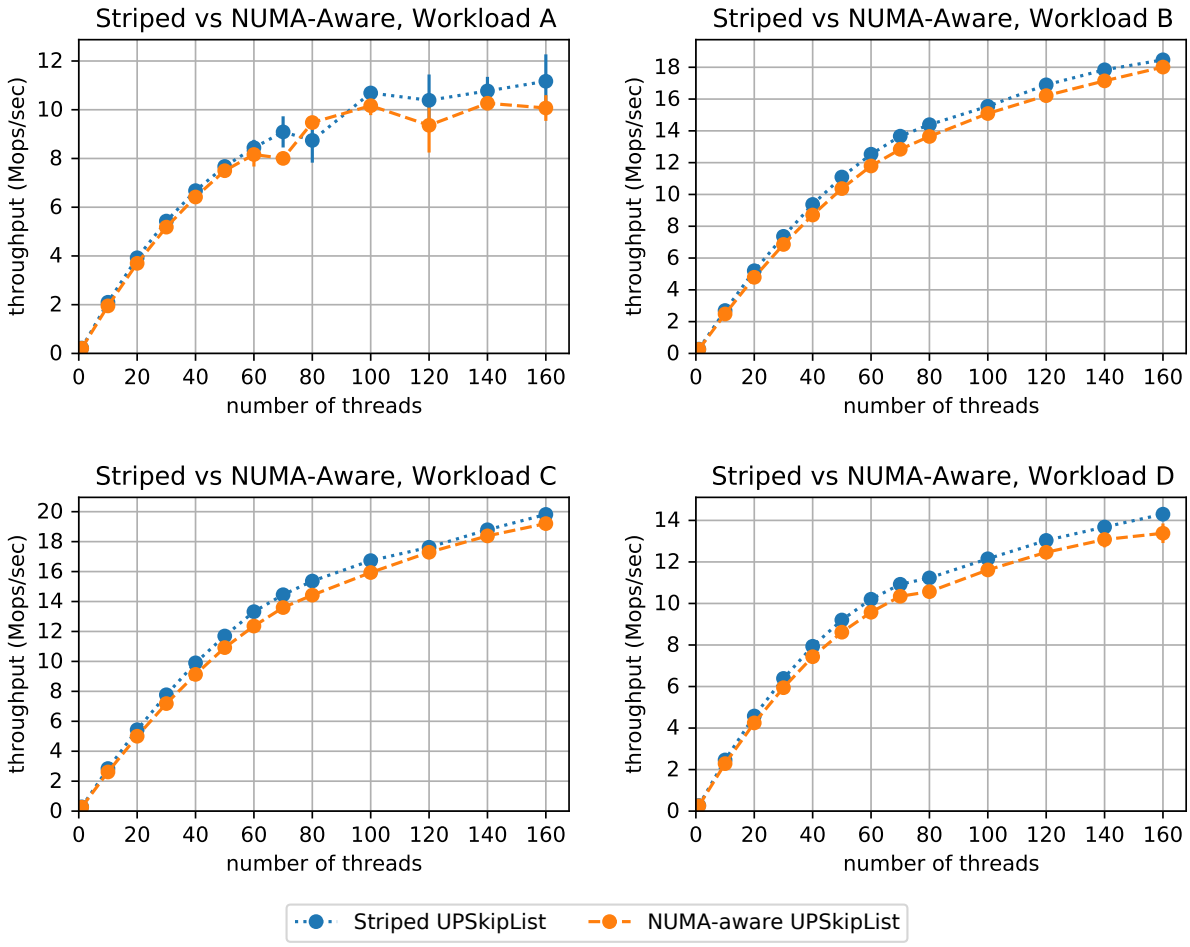


Figure 5.4: Throughput comparison of UPSkipList running on the striped device and on multiple pools.

### 5.2.4 Latency

Figures 5.5 and 5.6 show latency comparisons for UPSkipList and BzTree, and UPSkipList and the PMDK lock-based skip list, respectively. Table 5.3 shows the median latency in microseconds for each workload these data structures. Latency measurements were done with 80 threads for each workload, and separated by operation, to understand which operations were performing better than others.

For UPSkipList, in Figure 5.5, it can be seen that performance is very similar for update-heavy, read-mostly, and read-only workloads for reads, with minimal change in latency up to the 99th percentile, and 600$\mu s$ of latency at the 99.99th percentile. For the read-latest workload, it can be seen that the increased complexity of insert operations is impacting the latency of reads, due to the numerous flushes they require causing reads to have to fetch more cache lines from the memory. This results in a latency of 800$\mu s$ at the 99.99th percentile. The increased complexity of inserts is also exhibited in the latency of the inserts, which is higher for more operations compared to updates, with a noticeable increase in latency at the 99th percentile and a maximum of about 500$\mu s$ at the 99.99th percentile.

| Workload | Operation | UPSkipList | BzTree | PMDK skip list |
|---|---|---|---|---|
| Update-Heavy | Reads | 6.3 | 3.9 | 17.7 |
| Update-Heavy | Updates | 9.0 | 9.4 | 21.2 |
| Read-Mostly | Reads | 6.0 | 4.0 | 17.2 |
| Read-Mostly | Updates | 8.2 | 10.4 | 20.7 |
| Read-Only | Reads | 5.9 | 3.8 | 17.2 |
| Read-Latest | Reads | 7.0 | 4.2 | 18.2 |
| Read-Latest | Inserts | 12.1 | 16.4 | 36.3 |

Table 5.3: Median latency in microseconds for UPSkipList, BzTree, and the PMDK lock-based skip list for each YCSB workload.

Comparing UPSkipList to BzTree, it becomes clear why BzTree's performance suffers greatly in update-heavy and read-mostly workloads. While its abstraction using `PMwCAS` simplifies programming, it can be seen that update operations have up to an order of magnitude worse latency in the worst case, which is also more common. This result matches that of Lersch et al. [50]. The high latency of updates impacts read operations as well, with their performance clearly decreased in the update-heavy workload compared to the read-mostly workload, while UPSkipList's reads appear unaffected by the update percentage. Though the median latency of BzTree is shown to be lower than UPSkipList in Table

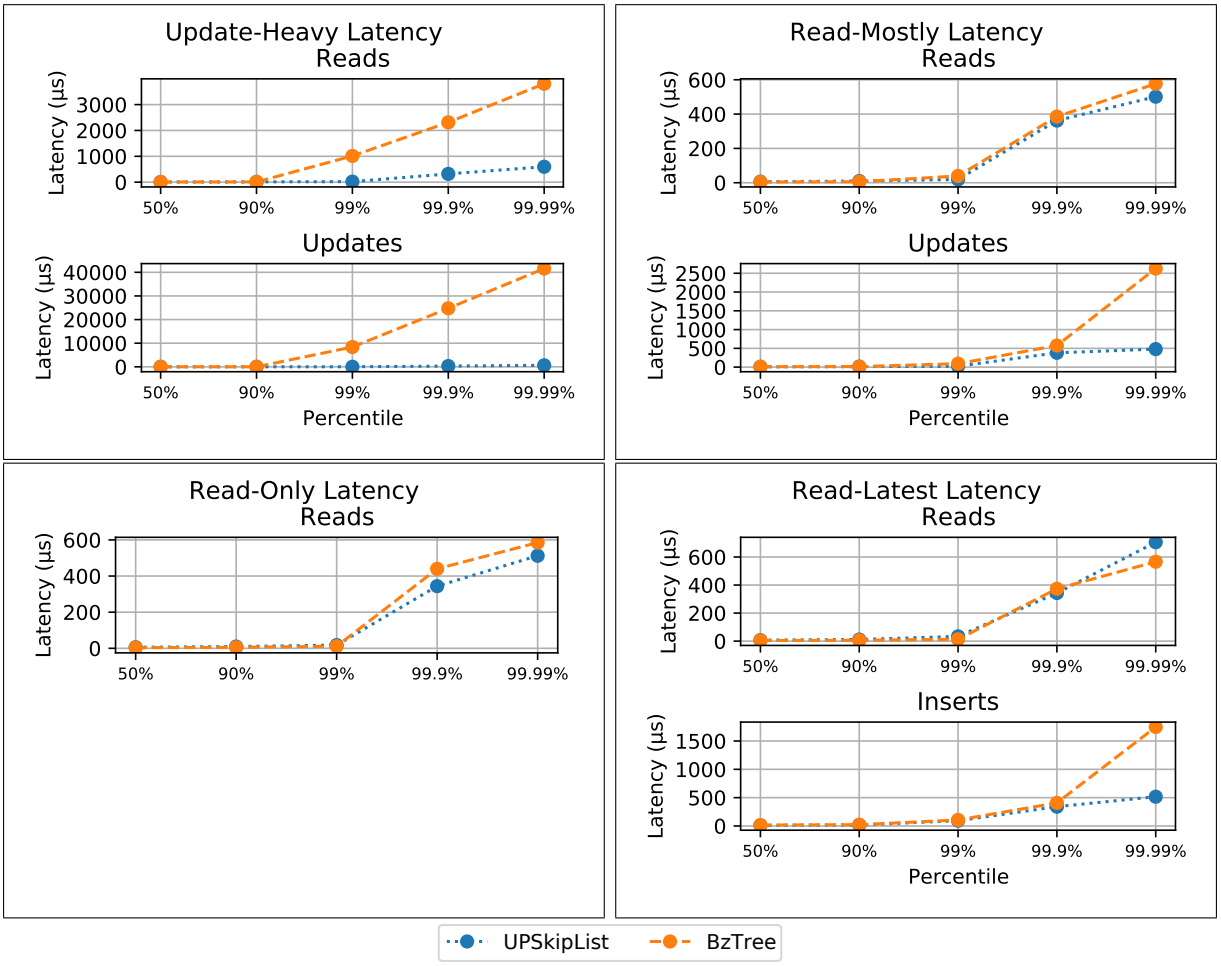


Figure 5.5: Latency at different percentiles for operations in each YCSB workload for UPSkipList and BzTree.

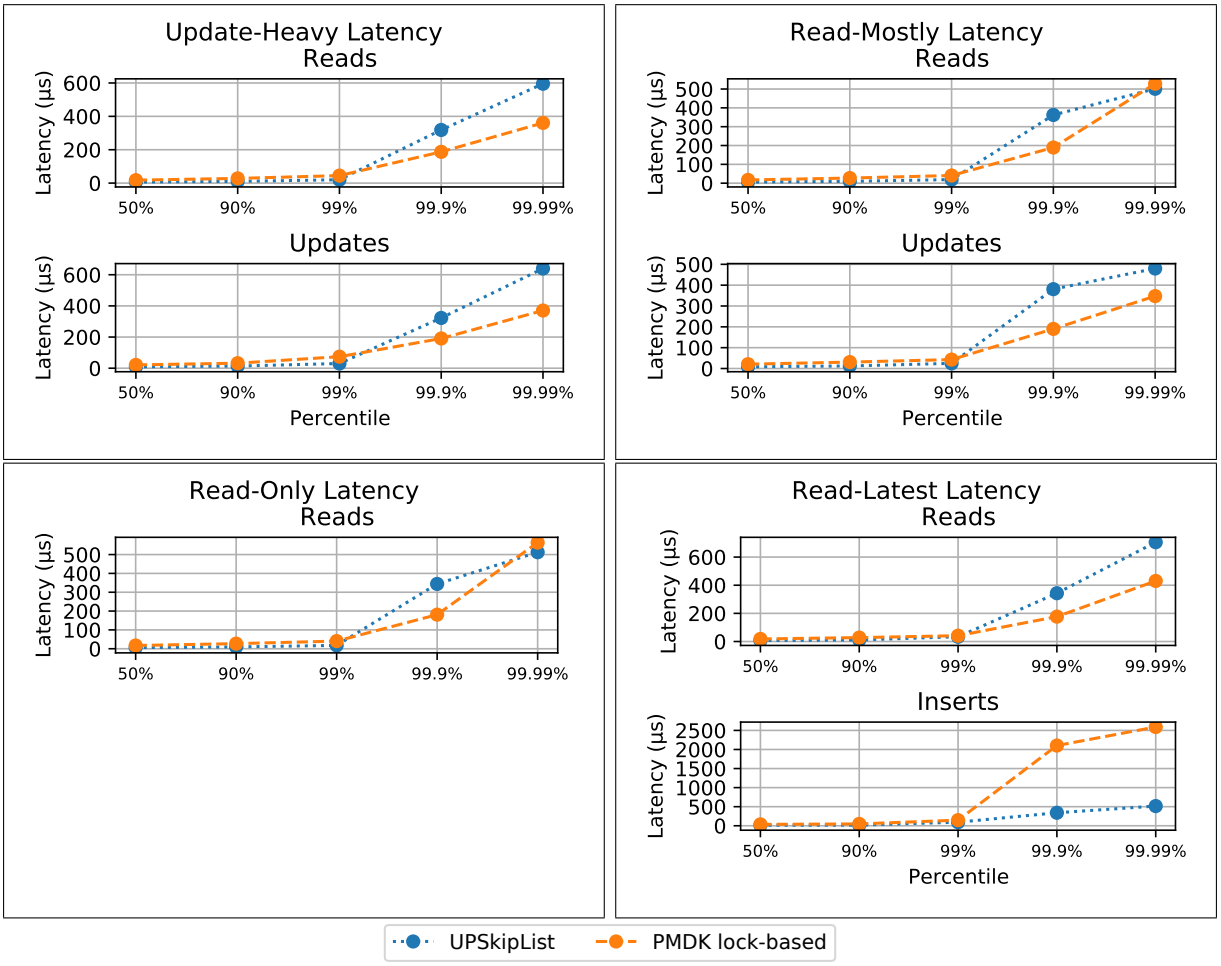


Figure 5.6: Latency at different percentiles for operations in each YCSB workload for UPSkipList and PMDK lock-based skip list.

5.3, this is outweighed by the increased likelihood of worst-case performance, as seen with BzTree's latency increasing from the 90th percentile. Examination of the BzTree codebase reveals that there is high contention on data structures internal to `PMwCAS`. The read-only workload latency is lower for BzTree than for UPSkipList, which is expected due to BzTree's better performance, as seen in Figure 5.2, and the lack of interfering updates or inserts. Although the worst-case latency is higher for BzTree in this workload, the median latency of reads, as seen in Table 5.3 reveals that the median latency is noticeably lower, explaining its improved performance. Finally, while inserts in the read-latest workload have higher latency for BzTree compared to UPSkipList, their impact on reads is much lower, which results in it outperforming UPSkipList for this workload.

The `libpmemobj`-based lock-based skip list's latency results are shown in Figure 5.6. While its performance is worse than UPSkipList for all workloads, the worst-case latency is not higher than UPSkipList's and is in fact lower for the update-heavy workload; rather the reduction in performance appears to be due to higher average latency, with a noticeable increase at both the 90th and 99th percentile compared to UPSkipList. Interestingly, the performance of its reads is also not greatly impacted by insert operations in the read-latest workload, showing the tradeoff of using lock-free insertions to improve performance at the expense of reads being invalidated.

### 5.2.5 Recovery Time

Recovery time measurements for all data structures are shown in Table 5.4. Measurements were done by running the test with a 100% insertion-based workload, which are more likely to be interrupted due to their complexity. The tests were run with 80 threads, preloaded with 100M keys, and interrupted after the workload began. Recovery time was then measured as the time required for the driver program to reconnect with each data structure's pool and perform all necessary preparation steps until it is ready to respond to new operations. The tests were run 3 times each, and the average recovery time is listed in the table below.

It can be seen that both UPSkipList and the `libpmemobj` lock-based skip list require a similar amount of time for recovery, with a sub-100 ms recovery time. This is expected, as both only have to reconnect to their pool, and `libpmemobj` has to roll back any incomplete transactions, of which there can be at most 80. The higher recovery time of UPSkipList is due to its less optimized reconnection procedure, compared to the lock-based skip list's reconnection being handled by `libpmemobj` itself.

BzTree, however, has a much longer recovery time than both UPSkipList and the

`libpmemobj`-based skip list. This appears to be due to the time needed to reinitialize the `PMwCAS` library, which is dependent on both the data structure size and the size of the `PMwCAS` descriptor pool. With 500K descriptors, which is the same number that all other tests in this chapter were run, recovery time is almost 10 times greater than both UPSkipList and the lock-based skip list. With 100K descriptors, which is the same amount used by Lersch et al. in their measurement of a recovery time of 186 ms, a recovery time of 239 ms is achieved, which is close enough to match their performance and verify that recovery time is dependent on the descriptor pool size. Unfortunately, running with only 100K descriptors results in the version of `PMwCAS` to constantly encounter garbage collection errors, which is the reason why the larger pool size was used for the rest of the testing. Recovery time for BzTree was not measured by the creators Arulraj et al. [5].

| Structure | UPSkipList | BzTree (500K desc.) | BzTree (100K desc.) | `libpmemobj` Lock-Based |
|---|---|---|---|---|
| Recovery time | 83.7 ms | 760 ms | 239 ms | 55.5 ms |

Table 5.4: Recovery time for each data structure, average of 3 trials

# Chapter 6

# Correctness

To detect and resolve any correctness issues in the implementation of UPSkipList's algorithm and recovery procedures, black-box linearizability testing was performed. Real power failures were used to crash the system and require UPSkipList to recover from inconsistent states. Analysis was done using the persistent synchronization primitive analyzer developed at the University of Waterloo [14], which checks for strict linearizability, necessitating its choice over alternative correctness conditions. Using the analyzer, 32 logs of UPSkipList with full power failures were analyzed and confirmed to be strictly linearizable.

## 6.1 Crash Testing

To be reasonably sure that UPSkipList is linearizable, crash tests were performed, logged, and analyzed. The tests were done on the same 80-core, 4-socket Intel Xeon Gold 6230 machine with 3072 GiB of persistent memory used for evaluating performance. The pre-crash portion of tests were done using a 100% insertion workload, both before and after the crash. This workload was used due to its insertion of new keys allowing the revelation of any linearizability errors within insertion, insert recovery, update, and read/traversal code, covering all possible points of error. This includes removals, due to their implementation being merely updating the value of a key with a tombstone.

### 6.1.1 Instrumentation

Due to the existence of full power failures, logging the start, end, and return values of operations to DRAM is not enough. Instead, logging was done using the `libpmemlog` library that is part of the PMDK [23]. Logging was done during both the preloading and runtime phases of the testing, so that initial values are known and can be checked using the analyzer to see if they are successfully found by later operations after a crash.

The analyzer requires that all written values be unique for each memory address so that correctness can be checked across a crash boundary in the logs. Uniqueness was ensured by using the logged start time of an operation as the insertion/update value.

### 6.1.2 Failure Injection

Linearizability analysis was done on logs generated during crash-free, simulated crash, and real power failure crash trials. All tests followed the same pattern of prepopulating the structure, running the workload either for a period of time or until a crash, reconnecting with the structure, running the workload again for a period of time, and then dumping the logs. Running the workload again after a crash causes it to update and re-read all of the keys that it had previously read or written to, allowing the analyzer to check them and determine if the correct values were found. To ensure that keys with interrupted operations will be written to after a crash, a small keyspace size of 50,000 keys was used. Crash-free tests served as a baseline, to ensure that without crashes the data structure appears to be consistent across multiple runs when shutdown cleanly.

Simulated crash tests were done by waiting a period of time after starting the workload, and then forcing the program to abnormally terminate by calling the `std::abort` function which sends a `SIGABRT` to the process tree. This prevents the program from finishing any operations in progress; however, cache lines will still be flushed by the kernel when unmapping the PMEM pool and be made durable before a clean shutdown or another execution of the program [21].

Full power failure crashes were performed to analyze the correctness of the data structure under the conditions which persistent memory and recoverable data structures are supposed to mitigate. By power cycling the server using its separate management module, the system does not have time to flush any cache lines that have not made it to the persistent domain, allowing for possible linearizability errors during recovery due to missing flushes in the implementation that would not be caught with simulated crashes.

While simulated crashes can be precisely timed, full power failures required the testing program to signal to an external machine that it is ready to be crashed, which is then initiated by the external machine by instructing the embedded Integrated Dell Remote Access Controller (iDRAC) module in the server to power cycle the server. Connecting to the external machine, initiating the power cycle, and the power cycle's occurence was found to require a variable amount of time between 3 and 10 seconds, causing the program to generate an excess of logs that take time to analyze. This was mitigated by signalling the crash to begin some time prior to the execution of the running phase of the test. Still, this variability resulted in logs varying from 20 MB to 400 MB, with logs larger than 50 MB being ignored because their large size is due to excess runtime prior to the crash that cannot contribute to revealing issues from interrupted operations.

## 6.2 Linearizability Analysis

The analyzer checks for strict linearizability of conditional swap operations by building a directed graph of all operations, mapping their dependencies on concurrent and prior operations and checking for cycles [14]. Crashes are handled by checking to see if interrupted operations appear to have taken effect, and if so, inserting responses for them with inferred values. Several other checks are performed as well to rule out other indications of nonlinearizability in the graph.

While this analyzer was chosen due to its support of crash operations, it was built to analyze logs containing CAS operations and not updates/inserts. Updates by UPSkipList can be treated as successful CAS operations by modifying them to return the previous value prior to the update, as UPSkipList uses CAS internally to update anyway until it succeeds. Similarly, inserts by UPSkipList can be treated as swapping out an initial value, which was arbitrarily chosen to be -1. Due to the use of the system time as the value being inserted, there is no chance of this breaking during the runtime of the test, barring changes to the system time. If the system time changes during a test, the results of that test can be ignored.

The analyzer performs analysis in $O(n \log n)$ time, and in practical use is effectively linear [14]. However, during testing it was found that it requires approximately 24 hours to analyze 150 MB of logs, while multiple gigabytes of logs are generated by 80 threads on a structure prepopulated with 100 million key-value pairs in less than a second. As a result, multiple changes were made to the test setup to reduce the log size to a manageable level. As previously mentioned, the key-value space was limited to 50,000 keys and only prepopulated to 20,000 keys, both limiting the number of keys that will be accessed and

increasing the likelihood of linearizability errors with threads more likely to access the same keys simultaneously. The number of threads was decreased from 80 to 20, which still proved to successfully detect linearizability errors. Finally, the tests were run with a target runtime of 100 milliseconds, though due to the unpredictable time to reboot the system the tests occasionally still reached several seconds in length.

## 6.3 Results

Initial tests run without failures, as a sanity check, were found to be linearizable. To first ensure that the analyzer will detect errors in the logs, logs were collected and manually modified to introduce linearizability errors by changing several read values at random. All of these errors were found by the analyzer and reported as nonlinearizable, which combined with the unit and integration testing done on the analyzer confirmed that it will successfully detect linearizability issues in logs containing crashes.

During the simulated crash tests, two linearizability errors were found by the analyzer revealing bugs in the implementation of UPSkipList's algorithm, again proving the effectiveness of the analyzer. The first error was found in the implementation of `DrainReaders()` used on line 122 in Function 10. It was using a write where it should have been using a compare-and-swap, allowing multiple threads to try to recover the same node, causing incorrect output. The second error found was in the testing program itself, where the values output by the program as `long` types were being truncated to `int` values occasionally. After fixing both these issues, the analyzer successfully found the simulated crash tests of UPSkipList to be linearizable.

For full power failure crash tests, the inability to precisely time failures meant that a much greater amount of tests had to be done to attempt to catch the program during a flawed execution. A script was run to repeatedly restart the machine from a second machine, following the procedure outlined in Section 6.1.2. The majority of the testing time went to analyzing linearizability logs, which depending on how long the server took to reboot took from around 10 minutes to 3 hours. Over night powercycle testing resulted in the generation and analysis of 32 logs. All logs were found to be linearizable, with no further issues detected.

# Chapter 7

# Conclusion

Due to the increasing dependence of society on technology and internet-based services, maintaining uptime without sacrificing performance is more important than ever. Using the extension to RECIPE described in this thesis, a new class of algorithms that are non-blocking and do not contain helping mechanisms can now easily be made recoverable, allowing persistent memory to help increase uptime in more applications. Performance can be further improved through effective persistent memory use in NUMA machines. By adding NUMA awareness to PMEM data structures beyond simply striping the storage pool across multiple nodes without drastically decreasing performance, the conversion and use of NUMA-aware algorithms in persistent memory is now practical. Using the extended Region-ID in Value method that uses the most significant bits beyond those used for addressing to allow referencing of memory in dynamically allocated objects, Recoverable memory allocation using the extended RIV method can also be deferred to the next attempted allocation of memory by a thread after recovery, reducing the time required until the servicing of new requests after a failure.

UPSkipList, a recoverable PMEM-resident skip list that is both NUMA-aware and implemented using the extension to RECIPE applied to Herlihy et al.'s lock-free skip list algorithm, shows the effectiveness of these techniques in building recoverable data structures in NUMA aware systems. Its performance beats that of BzTree in update-heavy workloads, where BzTree is bottle-necked by its use of PMwCAS for recoverability, rather than converting from an existing algorithm as was done with UPSkipList. UPSkipList also achieves comparable latency to BzTree when BzTree is not bottle-necked. The use of the extended RIV method of representing pointers shows performance improvements over using `libpmemobj` pointers due to improved cache efficiency. The extended RIV method also shows minimal overhead compared to striping the pool across multiple NUMA nodes,

opening the door to implementing fully NUMA-aware algorithms to achieve better NUMA locality in the future. In addition, the use of the RECIPE extension in implementation lends itself well to adding lock-free recoverable features to existing algorithms, with the implementation of multiple keys per node requiring just the addition of a way to check for inconsistency and a way to recover from this inconsistency when a node is found to be outdated. UPSkipList has been tested with interruptions due to full power failures to detect linearizability errors in its operation, with no errors detected across dozens of failures.

As next steps, for practicality reasons it is necessary to implement garbage collection in the extended RIV NUMA-aware memory allocation system, so that removes can be fully implemented and empty nodes can be reclaimed. Performance improvements can be achieved by implementing sorting of internal keys upon splitting a node, and performing a binary search upon these keys when performing searches. Linearizable range queries can be implemented to increase the usefulness of UPSkipList as an index data structure for databases. Testing UPSkipList against more recoverable data structures and in real database programs will give a better idea of how it compares to competition and the suitability its techniques for real-world applications beyond those tested using YCSB. Investigating the implementation of NUMA-aware techniques to improve locality of UPSkipList using multiple pools instead of striping data is another possibility for performance improvement. Overall, UPSkipList and the techniques used in its implementation show lots of potential for the future, with its contributions improving performance and reliability of data structures stored in persistent memory.

# References


[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical report, USA, 2003.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distrib. Comput.*, 13(2):99–125, April 2000.

[4] Marc Andreessen. Why software is eating the world. 2011.

[5] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, January 2018.

[6] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, page 299–308, New York, NY, USA, 2013. Association for Computing Machinery.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[8] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[9] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694, October 2016.

[10] Hadi Brais. Intel's clwb instruction invalidating cache lines, 2021.

[11] Trevor Alexander Brown. *Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way*, page 261–270. Association for Computing Machinery, New York, NY, USA, 2015.

[12] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. Technical report, USA, 2020.

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Diego Cepeda, Sakib Chowdhury, Nan Li, Raphael Lopez, Xinzhe Wang, and Wojciech Golab. Toward Linearizability Testing for Multi-Word Persistent Synchronization Primitives. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 191–203, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Qichen Chen, Hyojeong Lee, Yoonhee Kim, Heon Young Yeom, and Yongseok Son. Design and implementation of skiplist-based key-value store on non-volatile memory. *Cluster Computing*, 22(2):361–371, 2019.

[17] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[18] 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. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[19] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[20] C++ Reference Contributors. Low level memory management, 2018.

[21] Linux Kernel Contributors. mmap.c line 2680 - linux source code (v5.13.10), 2021.

[22] PMDK Contributors. *libpmemobj(7) man page*, 1.8 edition, 2020.

[23] PMDK Contributors. Persistent memory development kit, 2020.

[24] 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, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Intel Corporation. Persistent memory faq, 2020.

[26] T. Crain, V. Gramoli, and M. Raynal. No hot spot non-blocking skip list. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 196–205, 2013.

[27] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. Numask: High performance scalable skip list for numa. In *DISC*, 2018.

[28] 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, July 2018. USENIX Association.

[29] Tudor David and Igor Zablotchi. Concurrent lock-free data structures for non-volatile ram, 2017.

[30] Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4):e3876, 2017. e3876 cpe.3876.

[31] Doug Lea. Concurrentskiplistmap.

[32] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, page 50–59, New York, NY, USA, 2004. Association for Computing Machinery.

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

[34] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5–es, May 2007.

[35] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 377–392, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, page 65–74, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[38] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, page 265–279, Berlin, Heidelberg, 2002. Springer-Verlag.

[39] 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, page 3–16, Berlin, Heidelberg, 2005. Springer-Verlag.

[40] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, page 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[42] 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.

[43] K. Higuchi and T. Tsuji. A linear hashing enabling efficient retrieval for range queries. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, pages 4557–4562, 2009.

[44] Amazon Web Services Incorporated. What is a key-value database?, 2020.

[45] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. Linearizability of persistent memory objects under a full-system-crash failure model. volume 9888, pages 313–327, 09 2016.

[46] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[47] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Theo Leggett. Boeing 737 max lion air crash caused by series of failures. 2019.

[49] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proc. VLDB Endow.*, 13(4):574–587, December 2019.

[51] Chris Mellor. Cascade lake ap, optane persistent memory and endurance. 2019.

[52] 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, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery.

[53] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[54] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, page 224–238, Berlin, Heidelberg, 2013. Springer-Verlag.

[55] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. Concurrent unrolled skiplist. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1579–1589, 2019.

[56] William Pugh. Concurrent maintenance of skip lists. Technical report, USA, 1990.

[57] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[58] Andy Rudoff. Persistent memory programming. *USENIX ;login:*, 42(2):34–40, 2017.

[59] S. Thomas, R. Hayne, J. Pulaj, and H. Mendes. Using skip graphs for increased numa locality. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 157–166, 2020.

[60] Ticki. Skip lists: Done right, 2016.

[61] Mariano Trebino. Custom memory allocators in c++ to improve the performance of dynamic memory allocation, 2020.

[62] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[63] T. Wang, J. Levandoski, and P. Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

[64] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, 65(7):2169–2183, 2016.