Extremely fast (a,b)-trees at all contention levels

by

Anubhav Srivastava

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

© Anubhav Srivastava 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

Many concurrent dictionary implementations are designed and evaluated with only low-contention workloads in mind. This thesis presents several concurrent linearizable (a,b)-tree implementations with the overarching goal of performing well on *both* low- and high-contention workloads, and especially update-heavy workloads.

The OCC-ABtree uses optimistic concurrency control to achieve state-of-the-art low-contention performance. However, under high-contention, cache coherence traffic begins to affect its performance. This is addressed by replacing its test-and-compare-and-swap locks with MCS queue locks. The resulting MCS-ABtree scales well under both low- and high-contention workloads. This thesis also introduces two *coalescing*-based trees, the CoMCS-ABtree and the CoPub-ABtree, that achieve substantially better performance under high-contention by reordering and coalescing concurrent inserts and deletes. Comparing these algorithms against the state of the art in concurrent search trees, we find that the fastest algorithm, the CoPub-ABtree, outperforms the next fastest competitor by up to 2x.

This thesis then describes persistent versions of the four trees, whose implementations use fewer `sfence`s than a leading competitor (the FPTree). The persistent trees are proved to be strictly linearizable. Experimentally, the persistent trees are only slightly slower than their volatile counterparts, suggesting that they have great use as in-memory databases that need to be able to recover after a crash.

Acknowledgements

First and foremost, I would like to thank my supervisor Trevor for providing endless guidance support in our weekly “one hour” meetings that often stretched to four hours. I would also like to thank Wojciech Golab and Samer Al-Kiswany for agreeing to be on my committee, and for providing many useful comments.

As always, I cannot thank my family enough for their neverending love, support, and excellent food.

Finally, I would like to thank my fellow grad student Guy Coccimiglio for helping me understand persistent memory a little better, and my friend Nicholas Sunderland for reading early drafts of this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The (ordered) dictionary is one of the most fundamental abstract data types[1]. It stores a set of keys, each of which has an associated value, and provides operations to insert a key and value, remove a key, and find the value associated with a key. Dictionaries also support predecessor, successor, and range query operations. Dictionaries are often implemented using trees.

Concurrent dictionary implementations in the literature have typically focused on maximizing performance under low-contention workloads. These are workloads in which either most operations are read-only, or updates are performed on uniformly distributed keys (and thus rarely encounter contention). This thesis shows how to obtain high performance in update-heavy workloads with high contention, without sacrificing performance in read-heavy and low-contention update workloads.

The typical way to simulate high contention in concurrent data structure benchmarks is to use data structures containing only a small number of keys. In this case, a simple data structure (e.g. an array with locks for each element) is an acceptable solution. However, high contention can also occur when dealing with lots of keys. For example, in a data structure with many keys, a small subset of the keys might be updated very frequently. This can be simulated by using update-heavy workloads in which threads access keys according to a *skewed* probability distribution. In particular, this thesis focus on the Zipfian distribution, which is commonly used for simulating real-world access patterns [3, 5]. In the Zipfian distribution, the frequency of a key being accessed is proportional to its rank. That is, the kth most frequent key is requested with probability proportional to $1/k^s$,

[1]In this thesis, dictionary always refers to an ordered dictionary

where $s \geq 0$ is a parameter controlling the skew of the distribution ($s = 0$ is a uniform distribution, and the skew increases as s does).

Though this thesis primarily explores the Zipfian workload, the uniform access distribution is too important to ignore. Achieving good performance in the uniform workload is tied to the problem of minimizing cache misses within a search. Concurrent binary search trees (BSTs) are well-studied and have been heavily optimized for the uniform access distribution, but their performance still suffers because of their long path lengths: in a BST containing 1 million keys, searching for some keys will require traversing at least 19 nodes. The CPU's cache likely only contains nodes in the beginning of the path, since these nodes are accessed most frequently. This means that the rest of the accesses will incur cache misses and require fetching the nodes from a lower level of cache, or from main memory (a costly process). In fact, these cache misses dominate the runtime of BST operations.

In contrast to BSTs, B-trees use fat nodes, each of which contains many pointers. This results in B-trees having less depth than BSTs. A B-tree of order b has between $b/2$ and b pointers in each node, meaning that the height of the tree is at most $\lceil \log_{b/2} K \rceil$, where K of the number of keys in the tree. This can be significantly smaller than the $\lceil \log_2 K \rceil$ height of BSTs for large K, and practically leads to fewer cache misses and faster searching in B-trees. The time taken to search the keys of a B-tree node is negligible, since b is usually chosen such that a node fits in a single cache line. Moreover, rebalancing in B-trees occurs less frequently than in balanced BSTs. At first glance, it might not seem like a tree with fat nodes is the right data structure to use if one wants to achieve good performance under high contention. Although using fat nodes promotes better memory locality in low-contention workloads, it could easily become a liability under high contention, especially if multiple *hot keys* (which are updated frequently) are located in the same node. However, as shown in the experiments, fat nodes are crucial for obtaining high performance in low-contention and read-only workloads.

Cache misses are especially slow in architectures that feature non-uniform memory access (NUMA). NUMA architectures contain compute regions called **nodes** which share information via a network of buses/interconnections. NUMA architectures are becoming increasingly prevalent, as they have better performance than uniform memory access (UMA) architectures in workloads with high data access locality (the same data is accessed/modified by the same threads repeatedly). Threads within a NUMA node typically share some levels of cache. In the experiments in this thesis, the last level cache is shared amongst threads in a NUMA node. To read a memory address, a thread first checks some number of caches.

This thesis presents four data structure designs[2]. All four designs are based on (a,b)-trees, which are a generalization of B-trees that allow between a and b keys per node (where $a \leq b/2$). The first algorithm, the OCC-ABtree, uses optimistic concurrency control (OCC) to implement the relaxed (a,b)-tree of Larsen and Fagerberg [35]. OCC involves the use of **version numbers** to detect whether a node has changed. A node's version number is incremented at each update. Thus, if a thread reads a node's version number, does some local work, and reads the same version number again, it can be sure that the node was not updated between the two reads. The second algorithm, the MCS-ABtree, improves scaling compared to the OCC-ABtree under high contention by replacing test-and-compare-and-swap (TACAS) locks in the OCC-ABtree with MCS locks [42]. The third algorithm, the CoMCS-ABtree, adapts the idea of *elimination* from stacks for use in trees. In a stack with elimination, whenever a thread experiences contention while accessing the stack, it joins a group of threads that attempt to synchronize directly with one another to complete their operations *without* accessing the stack. The CoMCS-ABtree uses a technique called *coalescing* in which a thread that holds a lock traverses the MCS lock queue and communicates directly with other threads in the queue, enabling them to complete their operations *without* acquiring the lock. The proof of linearizability for this algorithm is subtle. The fourth algorithm, the CoPub-ABtree, introduces a faster coalescing technique that avoids the overhead associated with traversing lock queues.

Lastly, this thesis describes the changes required to make the aforementioned trees strictly-linearizable in systems with persistent main memory. Strict linearizability is similar to linearizability, with the additional requirement that operations cannot be completed after a crash. Given enough time, system failures such as power outages are inevitable. When a system failure occurs, any data that was not written to persistent storage (e.g. a hard drive) is lost. The persisted data can then be used to recover the data structure. However, even with modern persistent memory, writing every update to persistent storage is slow. Thus, programmers must carefully decide which data they want to persist. Chapter 7 of this thesis describes the modifications required to convert all the aforementioned trees into persistent trees, and experimentally examines the cost of persisting data and the benefits of coalescing in the persistent setting. The experiments show that the persistent memory trees are only slightly less performant than their volatile memory counterparts.

The main contributions of this thesis are as follows

- It presents four novel algorithms: OCC-ABtree, MCS-ABtree, CoMCS-ABtree and CoPub-ABtree, the last two of which include new techniques for *coalescing* in trees.

[2]The algorithms are publicly available, in the SetBench benchmarking suite: `https://gitlab.com/trbot86/setbench`.

All algorithms are shown to be linearizable and deadlock-free.

- The experiments performed cover a broad selection of competing algorithms, and the results show that the fastest algorithm in this thesis, the CoPub-ABtree, outperforms the next fastest competitor in a variety of workloads by up to 2x. Moreover, the CoPub-ABtree scales well under high contention, even with multiple sockets (tolerating the effects of non-uniform memory architectures), without sacrificing performance under low contention.
- It describes how to make all four data structures strictly-linearizable in a persistent memory system, in a way that requires very few `sfence` instruction (two for inserts which do not require splitting a leaf node and only one for deletes).
- It experimentally demonstrates that the persistent memory trees are almost as performant as their volatile memory counterparts in every workload (and thus coalescing is also practical in the persistent memory setting).

The remainder of this thesis is laid out as follows. Chapter 2 describes the theoretical models assumed by the algorithms in this thesis. Chapter 3 summarizes related work. Chapter 4 describes the OCC-ABtree, proves its linearizability and deadlock-freedom, and features a brief performance discussion showing that the use of MCS locks in the MCS-ABtree improves scaling across multiple processor sockets for high-contention workloads. Chapter 5 describes coalescing in the CoMCS-ABtree and CoPub-ABtree. Chapter 6 experimentally compares the OCC-ABtree, MCS-ABtree, CoMCS-ABtree, and CoPub-ABtree against a number of dictionary implementations on a variety of workloads. Chapter 7 describes the modifications necessary to make the above (a,b)-trees persistent and experimentally demonstrates that persistence does not incur too much overhead. Finally, Chapter 8 discusses possible future work and provides some concluding remarks.

Chapter 2

Model

The algorithms in Chapters 4 to 5 are developed for an asynchronous shared memory system. The algorithms in Chapter 7 target an asynchronous shared memory system with *persistent* main memory. This section begins by describing the asynchronous shared memory model, then describes the changes required for the persistent asynchronous shared memory model. Finally, we discuss additional considerations that affect the experiments in this thesis.

2.1 Asynchronous shared memory (ASM) model

2.1.1 Memory

The system's memory consists of a sequence of **words** which are indexed by their **address**. The word size is assumed to be large enough to fit the address of any word in the system.

The memory of the system can be divided into local and shared memory. Local memory is specific to a thread and can only be accessed or modified by that thread. Shared memory is shared amongst all threads and can be accessed and modified by any thread.

Threads access or modify memory using atomic memory primitives. There are four atomic memory primitives in this model: `read`, `write`, `swap`, and compare-and-swap (`CAS`). `read(x)` returns the contents of memory address `x`. `write(x,v)` stores value `v` in memory address x. `swap(x,v)` atomically stores value `v` at the memory address `x` and returns the value that was previously stored at `x`. Finally, `CAS(x,a,b)` atomically compares the word at address `x` with the value `a` and, if they are equal, stores the value `b` at address `x` and return `true`. Otherwise, `CAS` returns `false`.

2.1.2 Configurations, steps and executions

The **configuration** of the system is defined to be the contents of local memory (including each thread's program counter) and shared memory. In the ASM model, a **step** is a `read`, `write`, `SWAP`, `CAS`, or an invocation or response of a data structure operation.

An **execution** is a (possibly infinite) alternating sequence of configurations and steps: $C_0 s_1 C_1 s_2 C_2 \ldots$ (where C_i is a configuration and s_i is a step).

2.1.3 Correctness

Linearizability is a common correctness in the ASM model [31]. A concurrent execution α is linearizable if linearization points can be selected for each completed operation, and for a subset of the operations that started but did not complete, such that the linearization point for an operation occurs during the operation, and the result of each completed operation in α is the same as it would be if the operations were executed atomically at their linearization points. An algorithm is linearizable if every execution of that algorithm is linearizable.

2.2 Persistent asynchronous shared memory (PASM) model

The PASM model is intended to model the behaviour of crash-resilient systems with persistent main memory. In the PASM model, the system may randomly crash. Whenever a crash occurs, all threads crash simultaneously. The PASM model is similar to the ASM model; key differences are described below.

2.2.1 Memory

In the PASM model, we assume that there are two copies of memory: volatile memory and persistent memory. Threads only read from and write to volatile memory. Data can be copied from volatile to persistent memory; this is known as **flushing**. When a crash occurs, the contents of volatile memory are lost.

In addition to the memory primitives from the ASM model, the PASM model provides two new primitives, `flush` and `sfence`. Threads use `flush` and `sfence` to guarantee that their modifications have reached persistent memory.

`flush(x)`, also called an **explicit flush**, flushes memory address `x` when a subsequent `sfence` is executed by the same thread. In the PASM model, executing a write, a `flush` of the modified address, and an `sfence` guarantees that the write has been flushed to persistent memory by the time the `sfence` returns.

Memory addresses can also be flushed asynchronously without the programmer's knowledge (i.e. without calling `flush` or `sfence`). These are known as **background flushes**.

2.2.2 Recovery

Immediately after a crash, a **recovery procedure** is executed. The recovery procedure is often used to restore the data structure to a valid state (e.g. by completing or rolling back any partial updates). The recovery procedure has access to persisted memory. After the recovery procedure returns, a set of *new* threads resume operations on the data structure.

2.2.3 Configuration, steps, and executions

The configuration in the PASM model is defined to be the contents of both volatile and persistent memory. In the PASM model, a **step** is a `read`, `write`, `SWAP`, `CAS`, `flush`, `sfence`, background flush, crash event, or an invocation or response of a data structure operation. Background flushes and crash events are performed by a special system thread. This system thread exists solely for the model and does not correspond to a thread that would be running on a real system.

As in the ASM model, an **execution** is a (possibly infinite) alternating sequence of configurations and steps. The crash events in an execution partition the execution into segments called **eras**. Note that the ith era includes the recovery procedure after the $i-1$th crash (for $i > 1$).

2.2.4 Correctness

Linearizability was originally defined in a model which did not include crash events. Instead, the persistent algorithms in this thesis use **strict linearizability** as their correctness condition [4]. Strict linearizability is similar to linearizability, but forbids operations from linearizing after a crash. Strict linearizability also allows operations to abort mid-operation, but this feature is not required for the algorithms in this thesis.

2.3 System considerations

2.3.1 Caching

The system contains several **caches**, each of which contains a cached copy of a subset of the system's memory addresses. Data is added and removed from caches at the granularity of **cache lines**, which are fixed-size contiguous sequences of memory addresses. A cache line is 64 bytes and cache lines are 64-byte aligned. The caches are ordered in levels; L1 is the highest level of cache, followed by L2, and so on.

To access or modify data at a memory address, a thread starts at the highest level cache and works its way down until it finds the data. A thread is said to experience a **cache hit** if it finds the data in a cache. Otherwise, the thread is said to experience a **cache miss**. If the thread misses in every cache, it retrieves the data from main memory.

Higher levels of caches are smaller and shared amongst fewer threads than lower levels. For example, an L1 cache is typically shared between threads on the same core, whereas an L3 cache is typically shared between threads on the same processor socket.

The **cache coherence protocol** guarantees that all threads see the same view of the system's memory at any point in time. The protocol used in this thesis is the MESI protocol [47]. Modern processors implement more recent and more sophisticated protocols (such as MOESI, used by AMD [1]), but MESI suffices for the purposes of this thesis.

In MESI, a cache line is always in one of four states:

- Modified: the cache line has been modified by some thread since it was loaded into cache and is not present in any other cache.
- Exclusive: the cache line has *not* been modified by any thread since it was loaded into cache, and is not present in any other cache.
- Shared: the cache line has *not* been modified by any thread since it was loaded into cache, and may be present in other caches.
- Invalid: the cache line cannot be read from or written to (until it is read again from main memory).

To read a word from main memory or a lower-level cache, the cache line containing the word is loaded into all higher cache levels in the shared state. To modify the contents of a memory address, a thread must load the cache line in the exclusive state, which *invalidates* the cache line in all caches that the thread does not use. Cache lines are invalidated by sending a message to other caches though a bus. If more messages are being sent along the bus than it can handle (known as **cache coherence traffic**), the performance of the system can stall.

2.3.2 Non-uniform memory architectures (NUMAs)

We consider a NUMA system with multiple processor *sockets* (physical groupings of compute cores). Main memory is partitioned such that each socket has some *socket-local* memory associated with it, which can be accessed at a lower latency than the memory associated with other sockets.

Threads running on the same socket share the same last level cache, whereas threads running on different sockets do not. So, writes performed by one thread can be accessed by all other threads on the *same socket* via the last level cache. In contrast, threads on a different socket load the written data from main memory, which is typically much slower. As shown in Chapter 4, this can cause performance to *decrease* as the thread count increases.

2.3.3 Flushing

`flush`, as used in this thesis, corresponds to the `clwb` (cache line write back) x86 instruction. x86 also contains a `clflushopt` instruction, which is similar to `clwb` but invalidates the cache line, and a `clflush` instruction, which is equivalent to `clflushopt` followed by an `sfence`.

2.3.4 Allocation

We assume the existence of a memory allocator which provides an `allocate` operation to allocate memory for an object (e.g. a tree node) and a `free` operation which reclaims an object's memory.

To reclaim memory safely in a concurrent data structure, we also assume the existence of a *safe memory reclamation* algorithm which delays freeing objects until they are no longer reachable from the data structure, or from any thread's local memory.

In particular, the algorithms in this thesis use DEBRA [14], which is a fast implementation of epoch based reclamation (EBR).

Allocators in persistent memory systems must be carefully designed to avoid leaking memory when an allocation or deallocation is concurrent with a crash. Memory leaks are more severe issues in persistent memory systems than in volatile memory systems because leaked memory *remains* allocated (and hence leaked) after a crash.

In addition, the allocator should guarantee that pointers are recoverable after a crash (e.g. by guaranteeing that memory addresses do not change after a crash, or by providing the program with the new virtual memory offset associated with the process after a crash). This guarantee is required for the data structures in this thesis to be recoverable after a crash, since the recovery procedure follows pointers in the persisted tree nodes.

The persistent algorithms in Chapter 7 use jemalloc, which is not a persistent allocator and DEBRA, which is not a persistent memory reclamation scheme. If one wishes to actually implement these algorithms with a persistent allocator, one can use the Persistent Memory Development Kit (PMDK) software library, originally developed by Intel [52]. For persistent memory reclamation, NV-epochs from [20] is one possible option.

2.3.5 Scheduling and progress guarantees

This thesis uses the progress guarantees presented by Herlihy and Shavit in [30]. There are two possible progress guarantees for lock-based algorithms.

The weaker of the two is **deadlock-freedom**. Intuitively, deadlock-freedom states that *some* thread must continually complete operations in an infinite execution. Formally, an algorithm is deadlock-free if, assuming a fair scheduler (one that schedules each thread an infinite number of times):

1. In every infinite execution, an infinite number of operations return AND
2. There exists an infinite execution in which each thread returns from an infinite number of operations

The algorithms in this thesis are deadlock-free.

A stronger progress guarantee for lock-based algorithms is **starvation-freedom**. Intuitively, starvation-freedom states that *all* threads must continually complete operations in an infinite execution. Formally, an algorithm is starvation-free if, in every infinite execution, each operation returns in a finite number of steps (again assuming a fair scheduler). The algorithms in this thesis are not starvation-free. For example, a thread searching for a key can be prevented from returning by other threads executing insert and delete operations on that key.

Chapter 3

Related work

In this thesis, **external search trees** only store key-value pairs in the leaves. In constrast, **internal search trees** refer to search trees in which key-value pairs are stored in internal (non-leaf) nodes as well as leaves. The internal nodes of **partially external search trees** may or may not contain key-value pairs. Keys (in internal nodes) which are only used for directing searches to the correct leaf (and are not associated with a value) are called **routing keys** [9].

Binary search trees (BSTs). Ellen et al. introduced the first lock-free external BST [23]. Searches are implemented the same way as in a sequential BST. An update first searches for a *target* node to modify, then synchronizes with other updates by *flagging* or *marking* nodes to indicate how they will be modified. Updates that encounter these flags or marks will *help* the operation complete, guaranteeing lock-free progress. Natarajan and Mittal [44] improve upon this design by flagging/marking *edges* instead of nodes, and reducing the amount of memory allocated per update operation.

Bronson et al. [9] propose a partially external balanced BST (BCCO10) that uses optimistic concurrency control (OCC) to synchronize threads. OCC involves the use of version numbers (usually one per node, in node-based data structures like trees and linked lists) that are incremented at each update. If an operation reads a version number, performs some actions, and reads the same version number again, then no update modified the node between the reads. BCCO10 introduces a complex hand-over-hand version number based validation technique to implement fast searches. BCCO10 has previously been shown to be the fastest concurrent BST in search-dominated workloads [7]. The synchronization technique used in the OCC-ABtree for updates is somewhat similar to BCCO10, but searches are much simpler.

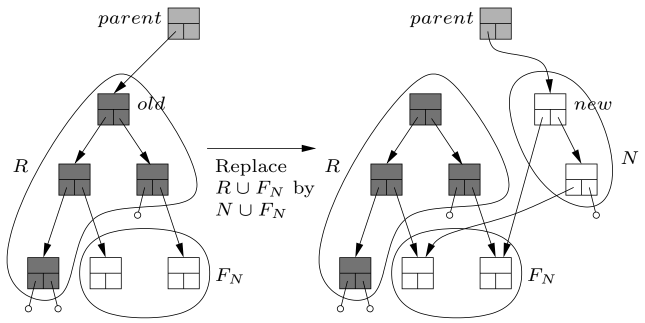

Figure 3.1: LLX and SCX primitives [12]

David et al. [21] propose a set of methods for optimizing concurrent data structures. The primary method is to optimistically construct an update locally and then atomically perform it on the data structure if the data used to construct the update has not changed. They use version numbers to detect changes in the data structure, as in optimistic concurrency control. They use this method to design a straightforward yet efficient lock-based external BST.

Brown et al. [11] introduced wait-free synchronization primitives (`LLX` and `SCX`), used them to implement a *template* for lock-free trees, and used the template to produce a (balanced) chromatic tree (a relaxation of a red-black tree) [12]. `LLX` and `SCX` are similar to the more well-known `LL` and `SC` primitives, but operate on *Data-records* instead of individual memory locations. A Data-record is a collection of *fields*. For node-based data structures, a Data-record usually represents a node. `LLX(v)` returns a snapshot of the fields of a Data-record `v`. An `SCX(D, F, fld, new)` by thread t atomically changes the value of a single field `fld` to `new` and marks every record in the set `F` as *finalized* (no longer able to be modified) only if none of the Data-records in `D` were modified since t's last `LLX` on each of them. Updates that follow the tree update template unlink a subgraph of nodes from the tree and link in a new subgraph to replace it (see Figure 3.1). Any nodes involved in the update are first read using `LLX`. Then, the thread performing the update constructs the new subgraph locally. Finally, the thread attempts to use `SCX` to link the new nodes into the tree and finalize any removed nodes. Although the algorithms in this thesis do not use the `LLX` and `SCX` primitives, they modify the tree structure similarly to the template (linking and unlinking subgraphs of the tree by only changing a single pointer in a node).

Several other concurrent BST algorithms have also been proposed [32, 45, 49].

B-tree variants. Brown used the aforementioned template to also design a lock-free external (a,b)-tree (referred to in this thesis as the LF-ABtree), which is a concurrency-friendly variant of an (a,b)-tree [10]. The LF-ABtree decouples structural operations (rebalancing the tree) from logical operations (insert, delete, and find) to increase performance. These structural operations were originally described by Larsen and Fagerberg in [35]. The LF-ABtree has been shown to be substantially faster than Natarajan and Mittal's trees and BCCO10, which are among the fastest BSTs [13]. The algorithms in this thesis improve upon the LF-ABtree by replacing the relatively slow read-copy-update approach for modifying leaves with in-place modifications. This involved replacing the `LLX` and `SCX` primitives with optimistic concurrency control techniques (version numbers) and modifying leaf nodes to use an *unsorted* array of keys. As the experiments in Chapter 6 show, the (a,b)-trees introduced in this thesis significantly outperform the LF-ABtree in many workloads.

The Bw-Tree is a lock-free variant of a B-tree that is designed to achieve high performance under realistic workloads [37]. It uses a mapping table to map logical nodes to physical pages. Each logical node contains a base state and a list of changes to it. Updates modify a record (e.g. insert a key-value pair) or perform a page operation (e.g. splitting a page) by adding to the list. This avoids the need to modify pages directly and hence reduces cache misses. The list of updates is periodically combined and applied to the base node. Many of the design decisions made in the Bw-Tree are focused on workloads that do not fit in memory, and incur significant overhead when the tree does fit in memory. The experiments in Chapter 6 include an optimized variant of the Bw-Tree called the OpenBw-Tree (since the original Bw-Tree is not publicly available) [58]. The optimizations introduced in the OpenBw-Tree increase performance by between 1.1x to 2.5x (depending on the workload).

The BzTree [8] simplifies the implementation of the Bw-Tree by using a multi-word compare-and-swap (MwCAS), and results in the paper suggest it is faster than the BwTree (though the official BzTree is not publicly available). Guerraroui et al. introduced a faster MwCAS algorithm and used it to accelerate the BzTree [56]. The BzTree can also be made persistent by using a persistent MwCAS. I attempted to compare with a public (unofficial) implementation of the BzTree, but encountered failures during validation [36]. The implementors mentioned that the errors might be fixable, but were unable to produce a fix.

In his thesis [41], McKenzie explores a number of modifications to a concurrent B-tree [39], with the goal of improving performance in systems with non-uniform memory access (NUMA). The most relevant modification to this thesis is the idea of using NUMA-aware locks. NUMA-aware locks improve performance in NUMA systems by preferentially passing the lock to threads on the same NUMA node as the thread currently holding the lock. This makes acquiring the lock faster. To maintain fairness between NUMA nodes, NUMA-aware locks often have a bound on the number of times that ownership of a lock can stay within a NUMA node (assuming some thread on another NUMA node wants to acquire the lock). The algorithms presented in this thesis could possibly benefit from NUMA-aware locks, but there may be a tradeoff with increased space overhead. McKenzie's thesis also explored include the replication of internal nodes (which is not applicable in the concurrent model used in this thesis), the ideal granularity of locking, and compared the performance of lock-free and locking reads. The author did not implement or test the delete operation (and so is excluded from the experiments in this thesis), but does sketch the changes necessary for it to be added.

Distribution/contention aware data structures. There has been also some work on data structures that are designed to accommodate non-uniform distributions.

The concurrent interpolation search tree (C-IST) of Brown et al. [13] uses internal nodes that are much larger than the internal nodes of most B-tree implementations: for example, the root node of the C-IST contains $O(\sqrt{n})$ keys. The C-IST uses linear interpolation to quickly find keys in internal nodes. This results in $O(\log \log n)$ runtime for smooth distributions (a set of distributions which includes the uniform distribution). The C-IST is primarily optimized for searching and, as shown in the experiments in Chapter 6, even a modest amount of updates can drastically reduce the C-IST's performance.

The splay tree [53] is a popular sequential dictionary that adapts to non-uniform distributions. After searching for a key, the splay tree performs rotations to move the node containing the key to the root. This reduces future access time for searches on the same key, but also introduces a point of contention at the root, which makes the splay tree unsuitable for concurrent use. The CBTree [3] is a concurrent splay tree-like data structure which uses counting to perform splaying only after a significant number of searches/updates have accessed a node, effectively amortizing the cost of a splay over many operations.

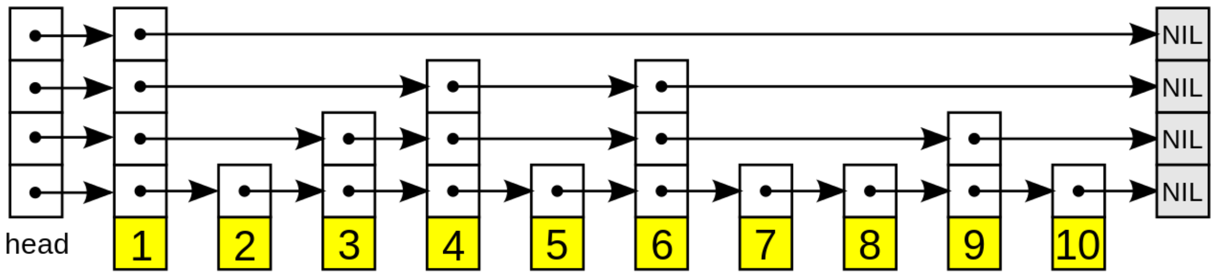

Figure 3.2: A skip list with height 4 [43]

The skip list [48] is a probabilistic dictionary with expected $O(\log n)$ searches and updates.

The skip list consists many stacked linked lists (see Figure 3.2). The lowest list contains all key-value pairs, and each higher list contains a subset of the key-value pairs in the layer beneath it. The height h of a key is defined by the highest list in which it appears. A search for a key in a skip list starts at the highest (and sparsest) list, and searches it until it either finds the key (and returns the associated value) or reaches a key after the search key. In the latter case, the search continues in the list below, starting at the last key read before overshooting. This continues until the search either finds the search key or does not find the key in the lowest list. An insert operation inserts the key-value pair in the lowest list and repeatedly flips a coin until it sees tails. For each heads it sees, it inserts the key-value pair in the next higher list. Delete operations remove a key-value pair from all linked lists it is in. Aksenov et al. introduced the Splay-List, a *concurrent* variant of a skip list that performs splaying by increasing the height of the most frequently accessed keys and decreasing the height of the least frequently accessed keys [5]. Like the CBTree, the Splay-List uses a counter-based approach to amortize the cost of the splaying. The authors of the Splay-List do not describe how keys should be deleted. The implications of this on performance are discussed further in Chapter 6.

The contention adapting search tree (CATree) [51] is a variant of an external binary search tree. Each leaf of the CATree is actually a sequential dictionary data structure, protected by a lock. AVL trees were used as the sequential dictionary in the authors' experiments (as well as in this thesis). When sufficient contention is detected at a leaf, the sequential data structure is split into two parts and an internal node connecting the two new leaves is linked into the tree. Similarly, two adjacent sequential data structures are combined if neither is under contention. The authors approximate contention at a leaf by measuring how often its lock is already acquired when a thread attempts to acquire it. Frequently-accessed keys are likely to appear as the only key in its leaf (i.e. sequential dictionary), which isolates other (neighbouring) keys from cache coherence effects. This is unlike B-trees, in which operations on keys sharing a leaf with a frequently-accessed key are likely to experience many cache misses. At first glance, having an isolated leaf for frequently-accessed keys appears to be optimal, but the coalescing technique introduced in this thesis achieves better than sequential performance on a single key.

General approaches. There are several universal constructions for transforming sequential data structures into concurrent ones. Though universal constructions are simple to use, they pay for their generality either by requiring multiple copies of the data structure (which is not practical for large data structures) or by using a global log or state object to serialize updates (which is a global bottleneck). [25] contains a summary of wait-free universal constructions. Of those wait-free universal contructions, there are five which do not require multiple copies of the data structure. The F-RedBlue and S-RedBlue constructions from [24] assume that the object can be modified by an LL/SC instruction, which is not true for a dictionary with more than a handful of elements. The SIM and P-SIM constructions from [25] and the construction from [18] have a global bottleneck when applying the operations. Lastly, Calciu et al. describe a lock-based universal construction that transforms sequential code into concurrent NUMA-aware code [16]. However, it uses a shared log, which represents a global bottleneck and also requires a copy of the data structure per NUMA node.

Transactional memory (TM) is another possibility for exploiting latent parallelism in sequential data structures. In TM, a *transaction* consists of a sequence of load and store instructions that either succeeds or fails atomically [27]. When a transaction reads (respectively, writes) a memory address, the address is added to the transaction's read set (respectively, write set). The union of a transaction's read and write sets is called its data set. If the data set of a transaction intersects the write set of a concurrent transaction, then one or both transactions will abort. This is known as a data conflict. Transactional memory can be implemented in hardware (HTM), software (STM), or a hybrid of the two (HyTM) [19]. HTM is usually more efficient than STM, but HTM implementations are usually best-effort: transactions may be aborted for reasons other than data conflicts (e.g. if the read or write sets grow too large for the hardware to handle) [28]. In contrast, STM is less efficient but can be run on systems without specialized hardware and typically offers progress guarantees. Regardless of how it is implemented, transactional memory is optimized for low-contention workloads. In the high-contention scenarios studied in this thesis, almost all transactions would abort because of data conflicts.

Persistent concurrent trees. Venkataraman et al. introduce criteria to create consistent durable data structures (CDDSs) [55]. These criteria are: durability and consistency in the event of a crash, scalability (minimal space, performance, and complexity overhead at arbitrarily large data structure sizes), and ease-of-use. They provide a B-tree which they claim satisfies the CDDS criteria. However, the pseudocode does not explain how threads are synchronized beyond mentioning a global version number that is incremented on each update (which is a scalability bottleneck) and does not appear to consider the effect of background flushes during updates (which I suspect can put the tree into an inconsistent state)[1].

Yang et al. created the NV-Tree, a persistent B-link tree (a B-tree in which nodes have sibling links to enable fast range queries) [60]. The NV-Tree, like the CDDS B-tree and

[1]Similar issues are raised by Wang in [57].

the algorithms presented in this thesis, uses unsorted leaf nodes. This enables fast insert and delete operations. The key difference between the NV-Tree and the persistent trees in Chapter 7 is that the NV-Tree rebuilds *all* of its internal nodes if *any* internal node becomes too full. As the authors mention, this can be extremely slow for large trees, but occurs less than 1% of the time in their workloads. Additionally, the NV-Tree only persists leaf nodes, since the entire tree can be recovered from them after a crash. This makes the recovery procedure slower, but avoids some flushes.

Unfortunately, it appears that the NV-Tree is not strictly-linearizable. When an insert into a full leaf node results in a split, a search may find the inserted key before the key is persisted. If a crash occurs before the pointer to the new node is persisted, the recovered data structure will not contain the inserted key. In this case, a search that sees the inserted key and returns before the crash cannot be linearized.

The Fingerprinting Persistent Tree (FPTree) is another persistent concurrent B-tree [46]. It includes a number of optimizations that make it scale better than the NV-Tree. Each leaf node includes a one-byte hash of each of its keys, known as a fingerprint. The fingerprints are scanned prior to probing the keys themselves, which limits the average number of key comparisons to 1. The authors note that this can have a large impact when key comparisons are costly (for example, if the keys are strings). Like the NV-Tree, the FPTree uses unsorted leaves and only persists leaf nodes (for fast insertion/deletion and minimizing flushing, respectively). Finally, the FPTree uses a combination of transactional memory (for internal nodes) and fine-grained locks (for leaf nodes) to synchronize threads. Using transactional memory for internal nodes simplifies the correctness argument for searches. Though it would be ideal to also use transactional memory for updates, transactional memory cannot be used in conjunction with persistence primitives. Thus, the FPTree's leaves have locks that are acquired by updates (and read by searches during their transaction to avoid seeing incomplete updates).

Chapter 7 presents persistent versions of the trees in this thesis. The new persistent trees are implemented using the link-and-persist method from [20] (a similar technique is proposed in [56]). Whenever new nodes are linked into the data structure, the pointer to the new nodes is marked to indicate that it is not persisted. The update then persists the pointer, removes the mark on it, and returns. Searches are accordingly modified to wait until a pointer is unmarked (and hence persisted) before following it. This guarantees that operations only access persisted data.

Chapter 4

A fast (a,b)-tree

4.1 Semantics

This chapter introduces a fast (a,b)-tree, the OCC-ABtree. The trees in this thesis all implement the following dictionary **operations**.

- `find(k)`: If a key-value pair with key `k` is present, return the associated value. Otherwise, return $\perp$.
- `insert(k, v)`: If a key-value pair with key `k` is present, return the associated value. Otherwise, insert the key-value pair `<k,v>` and return $\perp$.
- `delete(k)`: If a key-value pair with key `k` is present, delete it and return the associated value. Otherwise, return $\perp$.

Range queries could be added to these using the techniques described in [6], but are not described further in this thesis.

The OCC-ABtree also includes **rebalancing steps**, which are performed within operations but are not part of the data structure's interface. In this thesis, an **update** refers to either a rebalancing step or the insertion (resp. deletion) in an `insert` (resp. `delete`) operation. Note that an operation might contain several updates (e.g. an insert of a key, followed by two rebalancing steps).

The OCC-ABtree consists of an `entry` pointer to a sentinel node that is never removed. This sentinel node has no keys and just one child pointer, which points to the root of the tree. The pseudocode for the data structures used in the OCC-ABtree and its operations is presented below. To avoid cluttering the pseudocode, the memory reclamation operations are not shown. (However, the source code includes all of the necessary invocations.)

```
// K is key type, V is value type
abstract type Node
  marked    : bool
  lock      : TACASLock
  searchKey : K
  keys      : K[MAX_SIZE]

type Leaf inherits Node
  vals      : V[MAX_SIZE]
  ver       : int

type Internal inherits Node
  size      : int
  ptrs      : Node[MAX_SIZE]

type TaggedInternal inherits Internal

// The result of a search
type PathInfo
  gp     : Node // grandparent
  p      : Node // parent
  pIndex : int
  n      : Node // node
  nIndex : int

type RetCode is SUCCESS or FAILURE or RETRY

// Sentinel node: points to root
entry    : Internal
```

Figure 4.1: OCC-ABtree data structures

4.2 Data structures

The OCC-ABtree has three types of nodes: leaf nodes, internal nodes and tagged (internal) nodes. Leaf nodes store keys and values in their `keys` and `vals` arrays. We say an entry in the `keys` array is **empty** if it is $\perp$. An empty key has no associated value. Leaves contain at most `MAX_SIZE` keys and values. In the experiments, the minimum node size is 2 and the maximum node size is 11. A node is 232 bytes, and thus occupies 4 64-byte cache lines. The keys in a leaf are *unsorted* and there can be empty entries between keys.

Internal nodes contain `size` child pointers, and `size` $-$ 1 routing keys (that are used to guide searches to the appropriate leaf) in a *sorted* array. Once an internal node is created, its routing keys are never changed, but its child pointers *can* change. To add or remove a key in an internal node, one *replaces* the entire node. A new node is created to replace the internal node, and the parent of the internal node is updated to point to the new node. In contrast, keys in *leaves* are modified in-place (without creating a new node).

Conceptually, a tagged node (`TaggedInternal`) represents a temporary height imbalance in the tree. In a traditional B-tree, all leaves have the same depth. The OCC-ABtree satisfies a *relaxed* version of this balance property: all leaves have the same **relaxed depth**. The relaxed depth of a leaf is its depth, less the number of tagged ancestors it has. These imbalances represented by tagged nodes are gradually corrected using the `fixTagged` rebalancing step.

Each node also has a test-and-compare-and-set (TACAS) lock, and a node is only modified while it is locked. In a TACAS lock, the acquiring thread spins until the lock bit is unset, then attempts to acquire it with a compare-and-swap instruction. If it fails, it goes back to waiting until the lock bit is unset. TACAS locks are similar to test-and-test-and-set locks, and formative experiments suggest that there is no difference in their performance for the trees presented in this thesis. Leaf nodes have an additional version number field, `ver`, that records how many times the leaf has changed and whether it is currently being changed. After acquiring a leaf's lock, a thread increments the leaf's version before making any changes to the leaf and increments the version again once it has completed its changes, and finally releases the lock. **Thus, a leaf's version is even if the leaf is not being modified and odd if the leaf is being modified.** This is very important for the correctness arguments in this thesis. The version number is used by searches to determine whether any modifications occurred while reading the keys of a leaf[1]. Separating the version from the lock increases the window in which searches can read the leaf (and is also important for publishing coalescing in Chapter 5.2).

Nodes also contain a `searchKey` field that can be used to search for a node even if it does not contain any keys. The `searchKey` is set to an arbitrary key in the node when it is created, and is constant.

Finally, nodes contain a `marked` bit (initially `false`) which is set before a node is unlinked from the tree. Updates use the marked bit to quickly tell whether a node is in the tree. Once a node is marked, it is never unmarked.

Updates sometimes requires information about a node's ancestors. This is represented by the `PathInfo` structure. `PathInfo` is returned by `search` and contains the leaf at which the search terminated, the node's parent and grandparent, the index of the node in the parent's `ptrs` array, and the index of the parent in the grandparent's `ptrs` array.

4.3 Operations

All operations invoke a common `search` procedure, which takes a `key` as its argument, and searches the tree looking for `key` (starting at the root). At each internal node, `search` determines which child pointer it should follow by invoking `seqGetChild`, which traverses the node's routing keys sequentially. Once `search` reaches a leaf, it invokes `searchLeaf` to determine whether the leaf contains `key`. `search` returns `<rc, val, path>`. If `key` was

[1]A leaf's version field could hypothetically wrap around and cause an ABA problem, but at 100 million updates per second, this would take 2900 years for a 64-bit word size.

Algorithm 4.1: OCC-ABtree operations (search)

```
<RetCode, V> searchLeaf(leaf, key)
RETRY:
  ver1 = leaf.ver
  if ver1 is odd
    goto RETRY

  val = ⊥
  for keyIndex = 0 up to MAX_SIZE - 1
    if leaf.keys[keyIndex] = key
      val = leaf.vals[keyIndex]
      break
  ver2 = leaf.ver
  if ver1 ≠ ver2 goto RETRY
  if val = ⊥ return <FAILURE, ⊥>
  else return <SUCCESS, val>

int seqGetChild(node, key)
  numKeys = node.size - 1
  nIndex = 0
  while nIndex < numKeys and key ≥ node.keys[nIndex]
    nIndex++
  return nIndex

<RetCode, V, PathInfo> search(key)
  gp = NULL, p = NULL, pIndex = 0, n = entry, nIndex = 0
  while n is not Leaf
    gp = p, p = n, pIndex = nIndex
    nIndex = seqGetChild(n, key)
    n = n.ptrs[nIndex]

  path = PathInfo(gp, p, pIndex, n, nIndex)
  rc, val = searchLeaf(n, key)
  return <rc, val, path>

V find(key)
    rc, val, path = search(key)
    return val
```

Algorithm 4.2: OCC-ABtree operations (insert and delete)

```
V insert(key, val)
RETRY:
  rc, prev, path = search(key)
  if rc = SUCCESS return prev

  leaf = path.n
  parent = path.p
  Lock leaf
  if leaf marked
    Unlock leaf and goto RETRY

  // Verify key is not present
  for i = 0 to MAX_SIZE - 1
    if leaf.keys[i] = key
      Unlock leaf
      return leaf.vals[i]

  if leaf.size < MAX_SIZE
    // Insert without splitting
    for i = 0 to MAX_SIZE - 1
      if leaf.keys[i] = ⊥
        leaf.ver++
        leaf.vals[i] = val
        leaf.keys[i] = key
        leaf.size++
        leaf.ver++
        Unlock leaf and return ⊥
  else
    Lock parent
    if parent marked
      Unlock leaf/parent
      goto RETRY

    Mark leaf
    // Tagged if parent is not entry
    newNodes = Internal with two new
       children that evenly share
       contents of leaf and new key/val
    parent.ptrs[path.nIndex] = newNodes
    Unlock leaf/parent
    cleanup(newNodes.searchKey)
    return ⊥
```

```
V delete(key)
RETRY:
  rc, prev, path = search(key)
  if rc = FAILURE
    return ⊥

  leaf = path.n
  Lock leaf
  if leaf is marked
    goto RETRY

  for i = 0 to MAX_SIZE - 1
    if leaf.keys[i] = key
      deletedVal = leaf.vals[i]

      // Perform modification
      leaf.ver++
      leaf.keys[i] = ⊥
      leaf.size--
      leaf.ver++

      if leaf.size = MIN_SIZE - 1
        Unlock leaf
        cleanup(leaf.searchKey)
      else
        Unlock leaf
        return deletedVal
  return ⊥
```

Algorithm 4.3: OCC-ABtree operations (cleanup)

```
void cleanup(key)
RETRY:
  gp = NULL, p = entry, pIndex = 0
  do
    gp = p, p = n, pIndex = nIndex
    nIndex = seqGetChild(n, key)
    n = n.ptrs[nIndex]

    if n is TaggedInternal:
      fixTagged(gp, p, n, pIndex)
      goto RETRY

    if n.size < MIN_SIZE and n is not entry or root:
      fixUnderfull(gp, p, n, pIndex, nIndex)
      goto RETRY
  while n is not Leaf

fixTagged(gp, p, n, pIndex, nIndex)
  Lock n, p, and gp
  if n, p, or gp is marked
    Unlock all locks and return

  Mark n and p
  if p.size + 1 ≤ MAX_SIZE
    newNode = Internal combining keys/ptrs of n/p
    gp.ptrs[pIndex] = newNode
    Unlock all locks and return
  else
    // Tagged if parent is not entry
    newNodes = Internal with two children evenly sharing keys/ptrs of n/p
    gp.ptrs[pIndex] = newNodes
    Unlock all locks and return

fixUnderfull(gp, p, n, pIndex, nIndex)
  if nIndex = 0
    // Sibling is right neighbour
    sIndex = 1
  else
    // Sibling is left neighbour
    sIndex = nIndex - 1
  sib = p.ptrs[sIndex]

  if sib is TaggedInternal
    return

  // Lock leftmost sibling of n and sib first
  Lock n, sib, p, and gp
  if n, sib, p, or gp is marked
    Unlock all locks and return

  if n.size ≥ MIN_SIZE or n is entry or root
    Unlock all locks and return

  Mark n, sib, and p
  if n.size + sib.size ≤ 2 * MIN_SIZE
    // Distribute case
    newNodes = copy of p with pointers to two nodes sharing keys/ptrs of n/sib (without ptrs to n/sib)
    gp.ptrs[pIndex] = newNodes
  else
    // Combine case
    combined = New node with keys/ptrs of n/sib
    if gp = entry and p.size = 2
      // Remove p entirely
      entry.ptrs[0] = combined
    else
      newNodes = copy of p with pointer to combined instead of n and sib
      gp.ptrs[pIndex] = newNodes
  Unlock all locks
```

found, then `rc` is `SUCCESS` and `val` is the associated value. Otherwise, `rc` is `FAILURE` and `val` is ⊥. In either case, `path` is a `PathInfo` object as described in Chapter 4.2.

`searchLeaf` is similar to the classical double-collect snapshot algorithm [2]. It reads the leaf's version, reads its keys/values, then re-reads the leaf's version to verify that the leaf did not change while its keys/values were being read. If the leaf *did* change, then `searchLeaf` retries from scratch. If the key is found, `search` returns `<SUCCESS, val>`, otherwise, it returns `<FAILURE, ⊥>`. Note that `search` and `searchLeaf` do not acquire locks. This allows for greater concurrency, since (with some care) internal nodes can be updated *while* searches are traversing through them. Searches never restart from the root, unlike in other trees.

The `find(key)` operation essentially just invokes `search`.

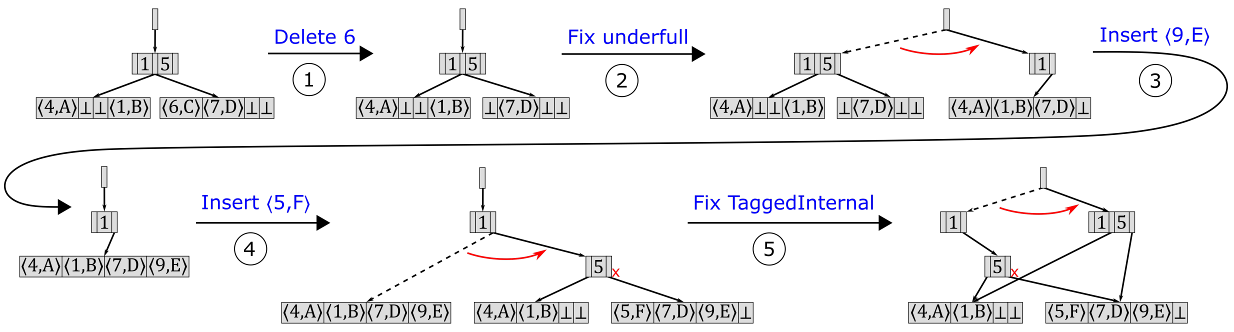

Figure 4.2: Operations on an OCC-ABtree with $a = 2, b = 4$. (1) The key-value pair ⟨6,C⟩ is deleted. This creates an underfull node. (2) The underfull node is merged with its sibling. This leaves the parent underfull, but the parent is the root, which is allowed to remain underfull. (3) ⟨9,E⟩ is inserted into an empty slot (*simple* insert). (4) No empty slot exists for ⟨5,F⟩, so the appropriate leaf is split, and a `TaggedInternal` node is created (splitting insert). (5) The `TaggedInternal` node is conceptually merged into its parent. We implement this by replacing its parent with a *new* `Internal` node.

In a `delete(key)` operation, a thread first invokes `search(key)`. If `search` does not find `key`, then `delete` returns ⊥. Otherwise, it locks the leaf and deletes the key by setting it to ⊥, and returns the associated value (Figure 4.2, 1). If `key` was deleted by another thread between `search` and acquiring the lock, `delete` returns ⊥. If deleting the key makes the node smaller than the minimum size, `delete` invokes `cleanup` to rebalance the tree with `fixUnderfull` (Figure 4.2, 2).

In an `insert(key, val)` operation, a thread first invokes `find(key)`. If `search` finds the `key`, then `insert` returns the associated value (Figure 4.2, 3). Otherwise, it locks the leaf

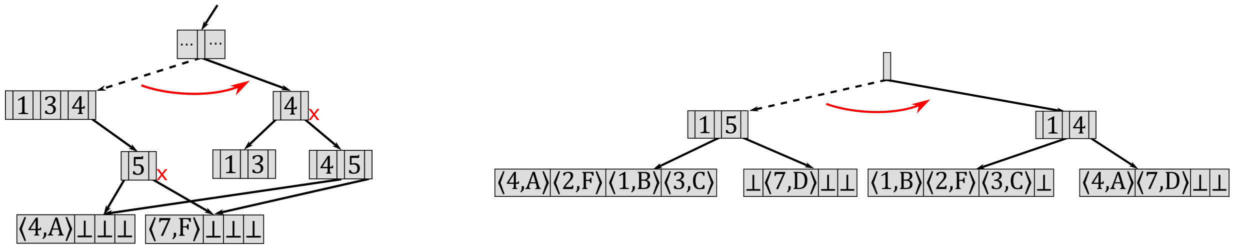

Figure 4.3: Left: `fixTagged` case where the parent of the tagged node is full and must be split (inserting a new tagged node). Right: `fixUnderfull` case where the siblings cannot be merged, so keys are distributed evenly instead.

and tries to insert `key` (resp. `val`) into an empty slot in the `keys` array (resp. `vals` array). We call this case a **simple insert**. If there is no empty slot, `insert` locks the leaf's parent and replaces the leaf with a new tagged node whose children contain the leaf's contents and the inserted key-value pair (Figure 4.2, 4). It then invokes `cleanup` to rebalance the tree with `fixTagged` (Figure 4.2, 5). We call this case a **splitting insert**.

`cleanup` searches for the key of a tagged or underfull node and attempts to remove any tagged or underfull node it encounters during its search. The fact that `cleanup` correctly rebalances the tree with its invocations of `fixTagged` and `fixUnderfull` is proved in Section 4.4.

`fixTagged` attempts to remove a tagged node. It tries to get rid of the tagged node by creating a copy c of its parent, with the tagged node's key and children merged into c, and changing the grandparent to point to c (Figure 4.2, 5). However, if the merged node would be larger than the maximum allowed size, `fixTagged` instead creates a new node p with two new children c_1 and c_2, which evenly share the contents of the old tagged node and its parent (Figure 4.3, Left). The grandparent is then changed to point to p. p is created as a tagged node, unless it is the new root, in which case it is simply an internal node.

`fixUnderfull` fixes a node that is smaller than the minimum allowed size by either merging it with a sibling node (if the contents of both nodes can fit in a single node) (Figure 4.2, 2) or by evenly dividing keys and values/pointers with a sibling node (Figure 4.3, Right).

4.4 Correctness

This section proves that the OCC-ABtree is linearizable. Recall that an algorithm is linearizable if, in every concurrent execution, every operation appears to happen *atomically* at some point between its invocation and its response.

Proving the linearizability of the OCC-ABtree requires a definition linking the *physical* representation of the OCC-ABtree (i.e. the contents of the system's memory) to the *abstract* dictionary it represents. The operations are then shown to modify the physical state of the tree in a way that is consistent with the abstract semantics described in Section 4.1.

4.4.1 Definitions

Definition 1 (Reachable node)**.** A node is said to be **reachable** if it can be reached by following child pointers from the entry node.

Definition 2 (Key in OCC-ABtree)**.** Let l be a reachable leaf. k is **in the OCC-ABtree** if, when l's version was last even, k was in l's keys array. Furthermore, if k is the ith key in l, the value associated with k is `l.vals[i]`.

In other words, a key is logically inserted or deleted when a thread increments the version number of the leaf for the second time (making it even).

Definition 2 is somewhat counter-intuitive. One might consider the following simpler definition: a key k is in the tree if it is in some leaf's keys array. Indeed, this alternate definition can also be used to prove that the OCC-ABtree is linearizable. However, Definition 2 is necessary for the correctness of publishing coalescing (Section 5.2). Using a consistent definition hopefully makes the correctness argument easier for the reader.

There are two more definitions which are used in the proofs throughout this thesis. The **key range** of a node is a half-open subset of the universe of keys (e.g. $[100, 200)$ if the keys are numbers, or [“*aardvark*”, “*apple*”) if the keys are strings). Intuitively, the key range of a node is the set of keys that are allowed to appear in the subtree rooted at that node.

Definition 3 (Key range)**.** The key range of the entry node is defined to be the universe of keys. Let n be a reachable internal node with key range $[L, R)$. If n has no keys, the key range of its child is also $[L, R)$. Otherwise, suppose n contains keys k_1 to k_m. The key range of n's leftmost child (pointed to by `n.ptrs[0]`) is $[L, k_1)$, the key range of n's rightmost child (pointed to by `n.ptrs[m]`) is $[k_m, R)$, and the key range of any middle child pointed to by `n.ptrs[i]` is $[k_i, k_{i+1})$.

Finally, the OCC-ABtree (along with all other trees in introduced in this thesis) is a relaxed (a,b)-tree, as introduced by Larsen and Fagerberg [35]. The relaxed (a,b)-tree is a search tree (as defined below). The most important consequence of the OCC-ABtree being a search tree is that, for any key k in the universe of keys, there is a unique search path for k, and this path passes through every reachable node whose key range contains k.

Intuitively, this path is the path an atomic search of k would take. Note that the uniqueness of the path implies that there is a unique reachable leaf in the search tree whose key range contains k.

Definition 4 (Search Tree)**.** Suppose n is an internal node in a tree and k is a key in n. A tree is a search tree if

- All keys in the subtrees to the left of k in n are less than k AND
- All keys in the subtrees to the right of k in n are greater than or equal to k

4.4.2 Invariants

Proving an `insert` is correct requires proving that `search` finds the correct leaf to insert into. For `search` to find the correct leaf, the tree must satisfy some structural properties, which are only satisfied if *previous* inserts and deletes were correct. This cyclical dependency is common in correctness proofs for concurrent algorithms. The standard way to deal with this is by assuming a set of invariants about the structure of the tree. These invariants hold for the initial state of the tree, and every modification to the tree preserves all invariants. These invariants can then be used to prove the linearizability of the data structure.

Theorem 5 (OCC-ABtree Invariants)**.** *The following invariants are true at every configuration in any execution of the OCC-ABtree.*

1. *The reachable nodes form a relaxed (a,b)-tree.*
2. *The key range of a node that was once reachable is constant.*
3. *A node that is not reachable contains the same keys and values that it contained when it was last reachable and unlocked (i.e. updates do not both unlink and modify a node).*
4. *A key appears at most once in a leaf.*
5. *If a node was once reachable, and is currently unmarked, it is still reachable.*
6. *If a node is unlocked and was once reachable, its* `size` *field matches the number of keys it contains.*
7. *The key range of* `n` *in* `search(key)` *contains* `key`*.*

Intuitively, invariants 1 to 4 follow from the sequential correctness of the updates together with the guarantee that any node that might be replaced or modified is locked and reachable until the update occurs. The sequential correctness of the updates (i.e. their correctness in a single-threaded execution) can be established by inspection of the pseudocode, so we do not prove it in detail. We briefly explain the (concurrent) correctness of invariants 1 to 4. Invariants 5 and 6 are straightforward from the pseudocode.

Invariant 7 is slightly different from the others, in that it is not a *structural* invariant. Rather, it describes the correctness of one of the operations. The proof is somewhat involved, so it is proved in detail.

Proof. The invariants hold at the initial state of the OCC-ABtree.

1: OCC-ABtree is a relaxed (a,b)-tree. The updates to the tree are the same as those described by Larsen and Fagerberg in [35]. They prove that, if these updates occur atomically, the tree is always a relaxed (a,b)-tree. Thus, the remainder of the proof simply shows that each update affects the tree atomically. This requires proving that for each update:

- There is a single step at which the update appears to take place
- The update is correct

The first condition is simple. Simple inserts and deletes appear when the modified leaf is unlocked, by Definition 2. All other updates only change a single pointer of a reachable node (to point to the update's newly created nodes).

For the second condition, assume that the updates are sequentially correct. This is easily verifiable by examining the pseudocode in this thesis and comparing it to the pseudocode in [35]. To establish concurrent correctness from sequential correctness, it is sufficient to show that the update occurs on the correct data (i.e. on the correct node and with any preconditions of the sequential code satisfied), the update affects data that is actually in the tree, and the data used to construct the update does not change while the update is being constructed.

An `insert(key, val)` operation uses the `search` function to find the leaf in which to insert. By invariant 7, the leaf's key range contains `key`. By invariant 1 (this invariant), the tree is a relaxed (a,b)-tree and thus a search tree, so there is a unique *reachable* leaf whose key range contains `key`. Finally, this leaf is reachable if the insert returns, because `insert` verifies that the leaf is not marked. Thus, the insert occurs in the correct leaf. A similar argument holds for `delete`.

The sequential code for the rebalancing steps has some preconditions. The `fixTagged` rebalancing step requires that the node is tagged, but its parent and grandparent node are not. This is guaranteed by `cleanup`. If any ancestor of the target node was tagged, cleanup would have invoked fixTagged on it and *restarted* the search. Also, fixTagged is only invoked on the target node if it is tagged.

fixUnderfull requires that none of the involved nodes are tagged, the parent node is not underfull, and the target node is underfull. The underfull node and its ancestors are not tagged, otherwise cleanup would have invoked fixTagged on them and restarted the search. The sibling node in fixUnderfull is also not tagged; this is explicitly verified. The parent node is not underfull, otherwise cleanup would have invoked fixUnderfull on it. Note that the parent could not *become* underfull because the keys of an internal node are constant. Finally, the target node is verified to be underfull. Thus, the rebalancing steps also act on the correct data.

Each update verifies that all involved nodes are not marked before performing its update. If the node is not marked, it is in the tree until the update itself unlinks the node, by invariant 5. Moreover, any children of the node are also in the tree by Definition 1. Thus, the data used to construct the update is actually in the tree.

Finally, the locks acquired by each update guarantees that any data involved in the update is constant until the locks are released.

2: Constant key range. We must examine places where existing nodes are attached to a new parent and ensure that the key range of all descendent nodes remains the same. This happens in fixTagged and fixUnderfull. In either function, the routing keys surrounding any pointer that is not removed remain the same before and after the update. Thus, the key range of the pointed to node does not change. This holds for leftmost and rightmost children of a node too, since the grandparent's key range does not change (by this invariant), and the new parent's key range is the same as the old parent's.

3: Unreachable nodes contain the same keys and values as they did when they were last reachable and unlocked. Updates that unlink a node first lock it, then unlink it, then unlock it, *without* changing the node's keys or values.

4: No duplicate key. Insert operations read the whole leaf while it is locked before attempting to insert a key, so a duplicate key is never inserted. The leaf that an insert operation tries to insert into is correct by invariant 7.

fixUnderfull does not create duplicate keys when merging two leaves because there is a unique leaf whose key range contains a given key, and any keys in that key range are only present in that leaf (invariant 1). Thus, a key can only be in one of the two leaves and so cannot appear twice in the merged node.

7: Search correctness. The search maintains the invariant that the key range of the node it is currently reading contains the search key. Call this node n. The invariant holds for the entry node, since its key range is the entire key space. Since the routing keys of an internal node partition its key range, there is a unique child whose key range contains the search key.

Let c be the child followed by the search after reading node n. Even if n is not in the tree when the pointer to c is read by the search, c must have been set as a child of n while n was in the tree, since only nodes in the tree are modified (invariant 3). Thus, at the time that n was in the tree and had c as its child, the key range of c contained the search key (Definition 3). Since the key range of a node is constant (invariant 2), the key range of c also contains the search key. □

With these invariants proved, the linearizability of the operations can now be established.

4.4.3 Linearizability of `find`

The leaf at which `search(key)` terminates was, at some point, the unique leaf that might have contained `key`, by invariant 7.

The search only returns if, during an interval when the leaf was unlocked, it either finds the search key and reads its value or it reads the entire leaf and does not find it. In the former case, we know that this key is unique in the leaf (invariant 4). Since the leaf was unlocked for the entire interval and nodes are not modified while they are unlocked, the result value of `find` is correct for the leaf state in that interval.

If the leaf was in the tree at any point in this interval, `find` may linearize at that point and be correct. If the leaf was never in the tree during the unlocked interval, `find` linearizes at the point just before the leaf was unlinked. The leaf must have been marked when it was unlinked; so, by invariant 3, the value returned by `find` is the same as it would be if the `find` occurred atomically just before the node was unlinked.

In this case, we must show that the `find` was concurrent with the point when the leaf was unlinked. Theorem 6 implies that the leaf must have been in the tree at some point during `find`'s invocation of `search`. Since (by assumption) the node was not in the tree in the unlocked interval, the `find` must have been concurrent with its unlinking. Note that the search procedure does not actually read the marked bit to check whether a leaf is in the tree; it is only described here for analysis.

Theorem 6. *Each node `search` visits was in the tree at some time during the search.*

Proof. The statement is true for the root. If the root is a leaf, the proof is complete. Otherwise, `search` reads a child pointer from the root. We now show that any child pointer read from a node n which was in the tree at some time during the search points to a child which was also in the tree at some time during the search.

If n is still in the tree at the time the child pointer is read, the child pointed to is also in the tree at that point by Definition 2. Thus the child is also in the tree at some point during the search.

Otherwise, n must have been (atomically) unlinked by some update U at time t. The search was concurrent with the unlinking of n since n was in the tree at some point during the search (by assumption) and n was not in the tree when the search read the child pointer. By invariant 3, the pointers of n point to its children just before it was unlinked at time t. Thus, the child followed by the search procedure was in the tree at some time during the search as well (namely, at t). □

4.4.4 Linearizability of `insert` and `delete`

There are four possible linearization points for an `insert(key, val)` operation. Note that in the final iteration of the `RETRY` loop, the leaf l that the `insert` locks is the *unique* reachable leaf that might contain `key` since the OCC-ABtree is a search tree (invariant 1), `key` is in l's key range (invariant 7), and l is not marked (invariant 5).

An `insert` that succeeds in its `search` is linearized in the same way as a `find` operation. The return value of the `search` is the value associated with the key (by the correctness of `find`) and is the correct value to return for `insert`.

An `insert` that finds `key` in the leaf l after acquiring the l's lock (and thus does not modify l) may linearize at any point while the l's lock is held because while l is locked, the key cannot be removed from l, the key's associated value cannot change, and l cannot be unlinked (since unlinking l would require marking it). Since the leaf's version is even, the associated value is the correct return value according to Definition 2.

An `insert` that inserts a key-value pair into a non-full leaf l linearizes at its second increment of l's version (which marks the modification as complete). The key is not in the OCC-ABtree *before* the linearization point since the insert read l while it was locked without finding the key, and l is the unique reachable leaf that might contain l. The key *is* in the OCC-ABtree after the linearization point (according to Definition 2) because the key is added to l, l is still reachable, and l's version is even.

For splitting inserts, searches can observe the change as soon as the pointer to the new subtree is written in the parent, since searches do not read locks on internal nodes. Thus, splitting inserts *must* linearize at the write to the parent node. Suppose a splitting insert writes the new pointer into the parent node p at time t. Let l be the leaf that was split and replaced by a tagged node t with children l_1 and l_2. The inserted key is not in the OCC-ABtree *before* the write to p since the insert reads l while it is locked and does not find the key (and l is the unique reachable leaf that might contain the key). *After* the write to p, the inserted key is in the tree because it is in either l_1 or l_2, both of which are reachable because p is unmarked and thus reachable (invariant 5). The other keys in l are not affected by splitting inserts since they are placed in one of l_1 or l_2 by the splitting insert.

The returned value of $\perp$ is correct in the above two cases, since the insert succeeded. The linearization of deletes and justification of return values is similar to the first three cases above.

4.5 Height bound

This section shows that the height of the OCC-ABtree is bounded by $O(c + \log_a n)$, where c is the number of threads currently executing an operation on the OCC-ABtree.

We now introduce the concept of a **violation**. A violation is a node which must be removed from the tree. There are two types of violations. A **tag violation** is any tagged node. An **underfull violation** is a non-tagged node, other than the entry node or root node, which is smaller than the minimum size. This sections shows that the number of *reachable* violations is bounded by c, then uses this bound to derive a height bound on the OCC-ABtree.

One more definition is needed before discussing the bound on the number of reachable violations. A **violating thread** is a thread that has created either a tag violation by splitting a leaf (in `insert`) or an underfull violation by making a leaf underfull (by deleting a key in it), and *has not returned* from its operation.

To show that the number of reachable violations in the tree is bounded by c, consider a mapping ρ from reachable violations to violating threads. Intuitively, if a reachable violation v maps to a violating thread T, T is responsible for removing v from the OCC-ABtree (by invoking `fixTagged` or `fixUnderfull` in `cleanup`). In some cases, T might also be responsible for removing new reachable violations created in the process of removing v. Table 4.1 describes when reachable violations are added to the mapping (by operations and rebalancing steps). A reachable violation v can be removed from the mapping in one of two ways (not described in the table): either v is unlinked from the tree (by a rebalancing step) or, if v is an underfull violation at a leaf, an insert makes the leaf not underfull.

Update type	Case	New entries in ρ
`insert`	Splitting insert, new tagged node t is not root	$\rho(t) =$ inserting thread
	Splitting insert, new node t is root	-
`delete`	Leaf n becomes underfull	$\rho(n) =$ deleting thread
`fixTagged` n with parent p	p not max size, new parent p' not underfull	-
	p not max size, new parent p' underfull	$\rho(p') = \rho(n)$
	p max size, new tagged node t (not root) with new children c_1 and c_2	$\rho(t) = \rho(n)$
	p max size, new node is root with new children c_1 and c_2	-
`fixUnderfull` n with sibling s and parent p	Combined node n' not underfull, new parent p' not underfull	-
	Combine: n' not underfull, p' underfull (and not root)	$\rho(p') = \rho(n)$
	Combine: n' not underfull, p' underfull (and is root)	-
	Combine: n' underfull, p' not underfull	$\rho(n') = \rho(n)$
	Combine: n' underfull, p' underfull (and not root)	$\rho(n') = \rho(n), \rho(p') = \rho(s)$
	Combine: n' underfull, p' underfull (and is root)	$\rho(n') = \rho(n)$
	Distribute keys, new parent p' not underfull	-
	Distribute: p' underfull	$\rho(p') = \rho(p)$

Table 4.1: Reachable violation to violating thread mapping

The bound on the number of reachable violations is a result of Theorem 7. The crux of the proof of Theorem 7 is Property 9, which guarantees that a violating thread is able to find the violation that maps to it by searching in `cleanup`. Thus, if a thread does not find any violation in its search, it can safely return.

Theorem 7. *The mapping described in Table 4.1 is injective.*

Proof. The mapping is initially empty, which matches the initial state of the tree.

Proving that the mapping is injective requires showing that each reachable violation maps to some violating thread, and no violating thread is mapped to by more than one reachable violation. We prove the latter statement first (since its proof is simpler).

Each violating thread is mapped to at most once. Let T be a violating thread. A reachable violation v is only mapped to T in one of two ways. The first way is if T created v through a splitting insert or delete operation. In this case, no reachable violation was previously mapped to T since T was not a violating thread before it created v. The second way is if, after a rebalancing step, a newly created reachable violation is mapped to T. Examining Table 4.1, it is clear that every update that does this also *unlinks* a reachable violation that previously mapped to T. Since the invariant guarantees at most one reachable violation mapped to T before the rebalancing, T still only has one reachable violation that maps to it after the rebalancing step.

Each reachable violation maps to some violating thread. A thread maps each new violation that it creates either to itself (in the case of splitting inserts or deletes that make a node underfull), or to a thread whose reachable violation it removed. In either case, the thread being mapped to is a violating thread, by the fact that the mapping is injective.

The only remaining way the theorem could be false is if a violating thread which is mapped to some reachable violation v returns. This would leave v mapped to a thread that is no longer a violating thread. Property 9 shows that this is never the case.

Property 9 implies that once a violating thread in `cleanup` reaches a leaf without encountering any violations, there are no reachable violations that map to the thread. Note that once a thread in `cleanup` has no reachable violation that maps to it, no *new* reachable violations will be mapped to the thread until it returns and performs a new operation (see Table 4.1). Thus, the thread may return without violating the invariant.

The proof of Invariant 9 relies on Property 8. The proof of Property 8 is by exhaustive examination of Table 4.1 and the sequential part of the rebalancing updates, and thus is omitted.

Property 8. *Let k be the* ***`searchKey`*** *of the violation originally created by a violating thread T. If a (not necessarily reachable) violation v maps to T, then the key range of v contains k.*

Consider a thread T performing a search in `cleanup`. The location of T, $loc(T)$ is the node pointed to by the pointer most recently read by T, or the entry node if T is at the beginning of `cleanup`. The **atomic completion** of T's search is the set of nodes that would be traversed along an atomic search for T's search key, starting at $loc(T)$.

Property 9. *While a violation v maps to a violating thread T and T is executing a search for key k in* ***`cleanup`****, one of the nodes in the atomic completion of T's search is a violation and T will not return from* ***`cleanup`****.*

Proof. The property holds whenever T is the entry node, since an atomic search from the entry node for k will pass through v (the OCC-ABtree is a search tree and the key range of v contains k, by Property 8).

The following two statements are sufficient to prove the property:

1. Every step of T's search maintains the property
2. Every update maintains the property

To see that the first statement is true, observe that T can only break the property by moving to another node or returning. If T moves to the entry node (because it invokes `fixTagged` or `fixUnderfull`) then the property is maintained (as argued above). Otherwise, if T moves to a child node from an internal node, then the internal node was not a violation (otherwise T would have fixed it). The child node T moves to is in the atomic completion of T's search because the keys of an internal node are constant (so T reads the same keys and child pointer that an atomic search would). Finally, suppose T returns after reaching leaf node l (which cannot be tagged). T did not call `fixUnderfull` and restart its search, thus it must have either read that `l.size > a`, l was the entry node, or l was the root. At that read, l is not an underfull violation. But, l is the only node left in the atomic completion of T's search (since l is a leaf). This contradicts the assumption that there *is* a violation in the atomic completion of T's search.

Every update also maintains the property. If an update does not unlink any node in the atomic completion, the atomic completion does not change, so the property is maintained. If an update does unlink a node in the atomic completion, there are two cases.

Case 1. Suppose there is a node n immediately before the unlinked node(s) in the atomic completion of T's search *and* n is reachable. Before the update, v was in n's subtree and thus was reachable. If v is not unlinked by the update, it is still reachable and in n's subtree after the update. If v *is* unlinked by the update, a new violation that maps to T will also be reachable and in n's subtree (if there is a new violation; otherwise no violation maps to T and it can return at any time). In either case, the fact that the OCC-ABtree is a search tree means that there is unique search path for k. This path will pass through n and then the violation mapping to T, since the key of both nodes contains k (Property 8). Since n is in the atomic completion of T, so is the violation.

Case 2. Otherwise, either there is no node in the atomic completion of T's search before the unlinked nodes, or the node immediately before the unlinked nodes is already unlinked. In the former case, since unlinked nodes are not changed when they are removed, the atomic completion remains the same. In the latter case, *every* node in the atomic completion from $loc(T)$ to the nodes being unlinked is not reachable (since, by definition, all children of a reachable node are reachable). None of these nodes can change because they are no longer reachable, so the atomic completion of T's search remains the same after the update. □

Thus, a violating thread does not return from `cleanup` until there is no violation that maps to it. Recall that no new violations map to the violating thread after this point (until it starts a new operation).

Thus, the mapping function is injective. □

The remainder of the height bound proof is relatively simple. Since the mapping function is injective, there can be at most c reachable violations at any time, where c is the number of violating threads. Consider removing each of these violations atomically. When a violation is removed by a rebalancing step, the height of the tree is reduced by at most one. Thus, removing c violations reduces the height of the tree by at most c. Since the OCC-ABtree is simply an (a,b)-tree (with height $O(\log_a n)$) when it contains no violations, its height with the violations is $O(c + \log_a n)$.

Theorem 10 (OCC-ABtree height bound)**.** *The height of the OCC-ABtree is $O(c+\log_a n)$, where c is the number of violating threads.*

4.6 Deadlock-freedom

Recall that an algorithm is deadlock-free if (assuming a fair scheduler),

1. In every infinite execution, an infinite number of operations return AND
2. There exists an infinite execution in which each thread returns from an infinite number of operations

Theorem 11. *The OCC-ABtree is deadlock-free.*

Proof. The second criterion is satisfied by considering an execution in which the scheduler picks a thread and performs its operation alone (without scheduling any other thread). This results in a sequential execution. Each operation will terminate in a finite number of steps because the operation's search will always reach a leaf node (invariant 1) and perform the desired operation. Thus, each thread returns an infinite number of times if the scheduler chooses each thread infinitely often (e.g. by cycling through threads in a round-robin way).

Now we prove that the first criterion is satisfied. Consider an infinite execution of the OCC-ABtree. Suppose for a contradiction that there exists a time t_1 after which no operations return. Let S be the set of operations that take steps forever.

Since no operations return after t_1, and each insert/delete only inserts/deletes at most once before returning, there exists a time $t_2 > t_1$ after which no key is inserted or deleted. After t_2, the only other updates that might modify the tree are rebalancing steps. Larsen and Fagerberg show that once inserts and deletes stop happening, only a *bounded* number of rebalancing steps can occur (Theorem 7 in [35]). Thus, there must exist a time $t_3 > t_2$ after which no rebalancing steps occur and therefore *the tree is no longer modified.* Since a version is set to odd only when the tree is about to be modified, all version numbers are even (and constant) after t_3.

To obtain a contradiction, we show that one of the operations in S must eventually return. Consider the following two cases.

Case 1: There is a `find` in S. No `find` can be stuck searching for a leaf forever since the OCC-ABtree is a search tree (invariant 1) that is not being modified, so the search will eventually reach a leaf. The search also cannot be stuck repeatedly executing the `RETRY` loop in `searchLeaf` because the leaf's version number is always even. So, `searchLeaf` should terminate and the `find` should return. This is a contradiction.

Case 2: There is no `find` in S. All operations in S are inserts or deletes. There are four places where an insert or delete might be stuck:

(a) The `RETRY` loop in `searchLeaf`

(b) The `RETRY` loop in `insert` or `delete`

(c) Waiting to acquire a lock that is held by another thread

(d) The `RETRY` loop in `cleanup`

We consider each of these subcases in turn.

Subcase 2a. `search` will eventually terminate for the reasons described in Case 1.

Subcase 2b. The operation cannot be stuck in the `RETRY` loop of `insert` or `delete` because it will eventually start a search after t_3 (when the tree is no longer being modified). Every node traversed in this search is linked in the tree, and is thus unmarked (invariant 5).

Subcase 2c. Suppose, after time t_3, some threads are each infinitely trying to acquire some lock. A thread only fails to acquire a lock if the lock is held by another thread. Consider the graph in which a vertex representing thread T has an edge to another thread T' if T' is holding the lock that T is trying to acquire. This graph must contain at least one cycle for threads to be stuck forever when trying to acquire a lock. The proof below eventually aims to prove that this cycle cannot exist without causing a contradiction.

Pick an arbitrary cycle and call the threads in this cycle T_1 through T_m. Each T_i has a lock on some node n_i and is trying to lock the node held by the thread after it in the cycle. Each of the T_i's can only be executing either a splitting insert, a `fixTagged` call, or a `fixUnderfull` call. Simple inserts and deletes acquire at most one lock, so they cannot be part of the cycle (they do not try to acquire a lock while they already hold some lock).

The proof that the cycle cannot exist uses the following definition. The **relaxed height** of a node in a relaxed (a,b)-tree (such as the OCC-ABtree) is the number of nodes between it and any leaf in its subtree (including one endpoint), *excluding* any tagged nodes [35]. The relaxed height of a node is well-defined: it is the same no matter which leaf in its subtree is chosen. For a node n, its relaxed height is written $rh(n)$. In the OCC-ABtree, the relaxed height of a node is *constant* once the node is linked into the tree. This is verifiable by examining the sequential pseudocode of each update.

The proof proceeds by showing that nodes are always locked in order of *non-decreasing* relaxed height. This implies that $rh(n_1) \leq rh(n_2) \leq \ldots rh(n_m) \leq rh(n_1)$. Thus, the relaxed heights of all n_i must be equal.

If T_1 (WLOG) is executing `fixTagged`, the node it is trying to lock (n_2) must either be the parent node or the grandparent node in its update (since it has already locked n_1). If n_2 is the parent node, $rh(n_2) = rh(n_1) + 1$ since n_2 is the parent of n_1 and is *not tagged* (`cleanup` guarantees there are no violations at the parent or grandparent). If n_2 is the grandparent, $rh(n_2)$ is either $rh(n_1) + 1$ (if n_1 is the parent) or $rh(n_1) + 2$ (if n_1 is the tagged node). Either way, $rh(n_1) < rh(n_2)$, which is a contradiction.

If T_1 (WLOG) is executing a splitting insert, n_2 is the parent node. If n_2 is *not* tagged, the contradiction is the same as in the `fixTagged` case above. Suppose n_2 *is* tagged. n_2's lock is held by thread T_2. T_2 cannot be executing a splitting insert, since n_2 would have to be the parent node of the splitting insert (n_2 is not a leaf). But then, T_2 would have acquired all of its locks, which contradicts the assumption that all threads are blocked. T_2 also cannot be executing `fixUnderfull` because no node involved in `fixUnderfull` is tagged (the sibling is explicitly verified, the other three are handled by `cleanup` calling `fixTagged` before `fixUnderfull`). Finally, as proven above, T_2 cannot be executing `fixTagged` as this leads to a contradiction.

The only remaining possibility is for *all* T_i to be executing `fixUnderfull`. If any T_i executing `fixUnderfull` is trying to lock the parent or grandparent node, the contradiction is the same as in `fixTagged`: $rh(n_i) < rh(n_{i+1})$. Otherwise, all T_i are locking the right sibling of their node n_i. But this implies that n_1 was once the left sibling of n_2, which was the left sibling of n_3, and so on until n_m was the left sibling of n_1. This is, even intuitively, a contradiction. Formally, suppose the key range of n_i is $[L_i, U_i)$. The key ranges of siblings are adjacent, so the above statement implies $U_1 = L_2$, $U_2 = L_3$, and so on until $U_m = L_1$. Thus, all L_i and U_i are the same. This is a contradiction because no node has an empty key range.

Thus, there cannot be such a cycle of threads T_1 to T_m. This implies that some thread will eventually acquire all of its locks and either return (e.g. if an insert finds the key it wishes to insert) or modify the tree (by inserting/deleting a key or executing a rebalancing step) after t_3, which is a contradiction.

Subcase 2d. Finally, suppose none of the other subcases occurs. Thus, *every* thread is infinitely searching in `cleanup`. A thread T will eventually reach a violation v at some time after t_3 since it searches forever and returns if it does not find a violation.

If v is a tag violation, T will acquire all locks and fix the violation (contradicting the fact that the tree is no longer being modified after t_3). T will not retry after acquiring the locks because no search after t_3 reaches a marked node. Note that T *will* eventually acquire all locks because it performs an *infinite* number of searches, so it must *eventually* acquire all locks in order to continue searching.

If v is an underfull violation, T will call `fixUnderfull`. If the sibling node is not tagged, T will eventually acquire all locks and fix the violation, which is a contradiction. If the sibling node *is* tagged, the injective mapping from Section 4.5 guarantees that there is *some* violating thread (call it T') to which v is mapped. T' will reach v because it is infinitely searching in `cleanup` and there are no violations on the path to v (otherwise T would have fixed them). Once T' reaches v, it will call `fixTagged`, successfully acquire all of its locks, and fix the violation. Again, this is a contradiction. □

The other trees in this thesis are also deadlock-free. The proofs are similar, and so are omitted.

4.7 Performance vs lock-free implementation

The OCC-ABtree is based on the lock-free (a,b)-tree (LF-ABtree) of [10], which often outperforms the state-of-the-art in binary search trees. The LF-ABtree uses the wait-free primitives `LLX` and `SCX` (extended versions of the well-known `LL` and `SC` primitives). A key is inserted by replacing a leaf with a new copy that contains the new key. This read-copy-update style of insertion is encouraged by the `LLX` and `SCX` primitives (since `SCX` can only modify a single pointer atomically). Copying every key and pointer/value in a node every time it has to be updated adds significant overhead to updates, so the LF-ABtree does not scale well in heavy-update workloads. This is true even if the updates are uniformly distributed. In contrast, the OCC-ABtree modifies leaves in place, leading to more efficient updates.

We briefly compare the performance of the OCC-ABtree and the LF-ABtree to motivate further algorithmic improvements. In these experiments, n threads all access the same tree containing 1 million keys (and values) for 10 seconds. The experiments in this chapter were run on a 4-socket Intel Xeon Gold 5220 with 18 cores per socket and 2 hyperthreads (HTs) per core, for a total of 144 hardware threads, and 192GiB of RAM. In all experiments, threads are pinned such that the first socket is saturated before the second socket is used, and so on. Additionally, the pinning ensures that all cores on a socket are used before hyperthreading was engaged (i.e. hyperthreading is engaged for all thread counts other than 18 threads, in the graphs below). The full experimental setup is described in detail in Chapter 6. We test three workloads: search-only with uniformly distributed accesses to keys in the data structure, update-only with uniform accesses, and update-only with Zipfian accesses.

The OCC-ABtree and LF-ABtree perform similarly in the search-only workload (Figure 4.4), but the OCC-ABtree outperforms the LF-ABtree on the uniformly distributed, update-only workload (Figure 4.5) because of the reduced update overhead. In the update-only Zipfian workload (Figure 4.6), the LF-ABtree has much better scaling, but its peak

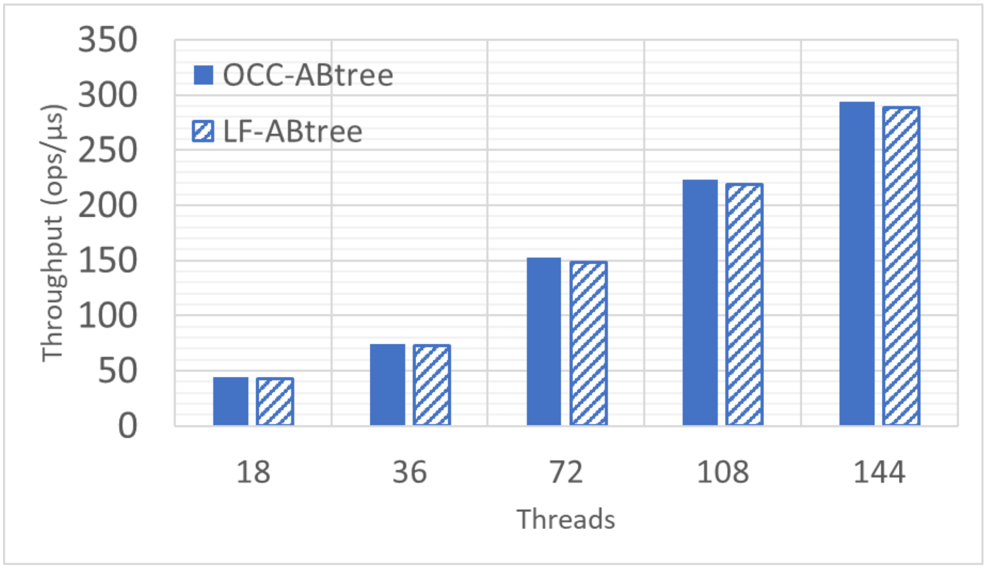

Figure 4.4: Throughput comparison of the OCC-ABtree and LF-ABtree on a uniform-access, search-only workload.

performance is only marginally better than the OCC-ABtree. The poor scaling of the OCC-ABtree under high-contention workloads is a result of contention on the TACAS lock of the few leaves containing frequently updated keys. When an update completes, any concurrent updaters that were waiting on the lock rush to acquire it, causing significant cache coherence traffic. This causes *stalled cycles*. This performance problem is magnified as the number of processor sockets increases.

4.8 Using MCS locks

As a first step towards mitigating this performance problem, we replace the TACAS locks in the OCC-ABtree with MCS locks [42]. These are queue-based locks which are ideal for high-contention, since they offer constant remote memory reference complexity. A thread attempting to acquire a lock creates a *queue node*, and joins a queue of threads waiting to acquire the lock, spinning until the `owned` bit in its queue thread is set to `true` by the thread just before it in the queue. Spinning locally instead of on a heavily contended bit eliminates the problem of multiple threads rushing to acquire the lock as soon as it is released. As an added bonus, MCS locks are fair, which may have some beneficial effect in practice on the tail latencies of updates. The only change to the OCC-ABtree pseudocode

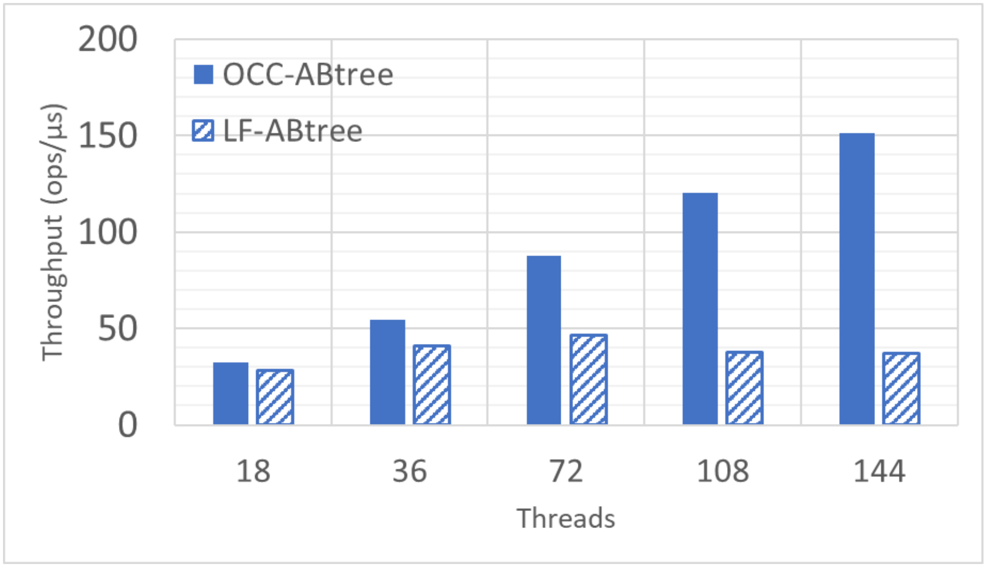

Figure 4.5: Throughput comparison of the OCC-ABtree and LF-ABtree on a uniform-access, update-only workload.

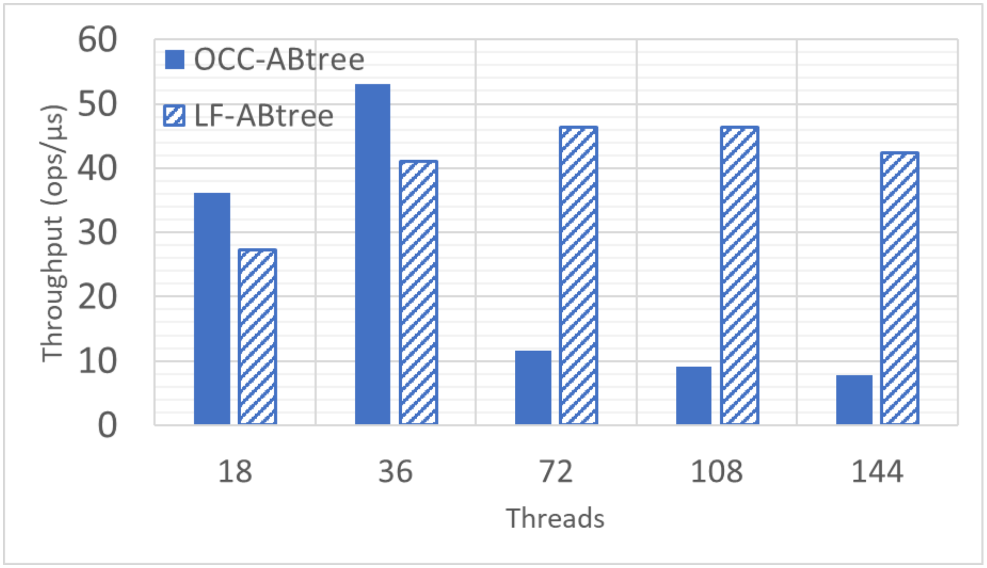

Figure 4.6: Throughput comparison of the OCC-ABtree and LF-ABtree on a Zipfian-access, update-only workload.

is the replacement of the TACAS lock on line 4 with an MCS lock.

The space overhead of using MCS locks is minimal. Each node only needs a pointer-sized field to point to the tail of the MCS queue, and each thread only ever needs four MCS queue nodes, since no operation acquires more than four locks at a time. Note that these queue nodes do not need to be allocated on the heap. (In the experiments in this thesis, they are stack allocated, which is highly efficient.)

Figures 4.7 to 4.9 illustrate the effectiveness of using MCS locks in workloads with high contention. Figure 4.7 is the same experiment as in Figure 4.6 but with the MCS-ABtree added in. Figure 4.8 compares the average number of cycles per operation spent stalled while waiting for cache coherence on an update-only, Zipfian workload (as measured by the `PAPI_RES_STL` performance counter in the PAPI C++ library [54]). The hardware setup is as described in Chapter 6. Finally, Figure 4.9 compares the average time taken to acquire the lock. Note the logarithmic scale on Figures 4.8 and 4.9: the OCC-ABtree spends roughly *an order of magnitude* more time stalled and trying to acquire the lock when running on multiple sockets (compared to both MCS-ABtree and its own single socket performance).

The degree to which the use of MCS locks addresses the scaling issues of the other two algorithms is quite surprising. It would be interesting to investigate whether similar gains could be made in other lock-based concurrent data structures with this simple change.

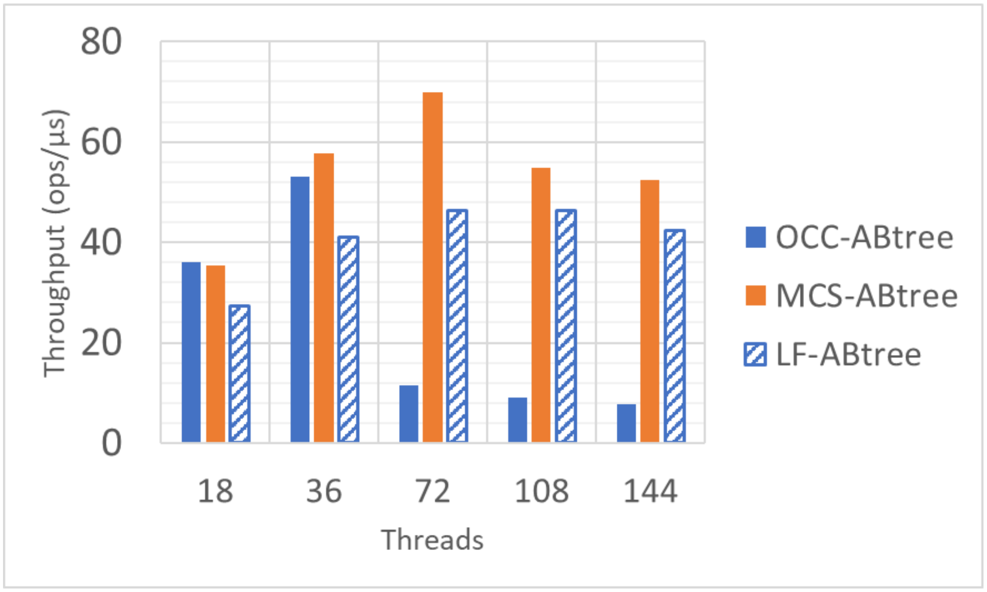

Figure 4.7: Throughput comparison of the OCC-ABtree, MCS-ABtree, and LF-ABtree on a Zipfian-access, update-only workload.

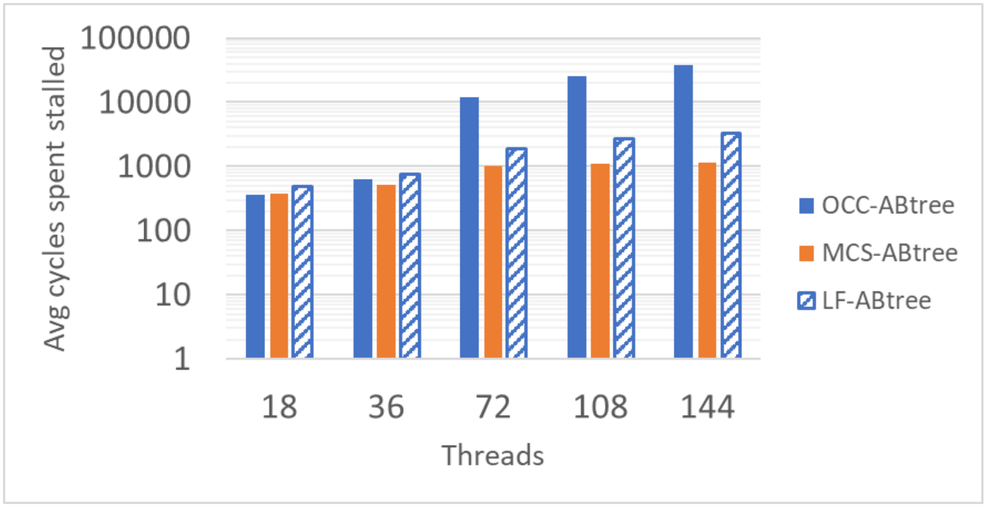

Figure 4.8: Average number of cycles the OCC-ABtree, MCS-ABtree, and LF-ABtree are stalled when running a Zipfian-access, update-only workload.

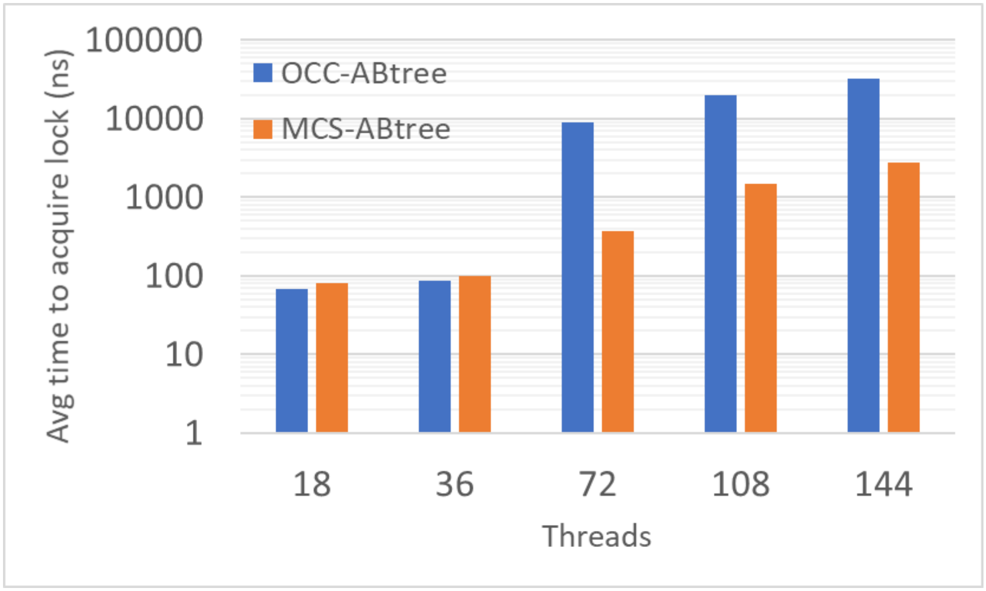

Figure 4.9: Average time a thread in the OCC-ABtree and MCS-ABtree takes to acquire the lock, when running a Zipfian-access, update-only workload.

Chapter 5

Coalescing

Although MCS locks substantially improve performance, there is still a scaling bottleneck caused by the serialization of modifications to *hot* nodes (which contain frequently updated keys). This problem is reminiscent of the scaling problems faced by designers of concurrent stacks and queues. In this section, we adapt *elimination*, one of the most impactful algorithmic innovations in concurrent stacks, for use in trees.

Elimination in a stack reduces contention by trying to pair up concurrent push and pop operations, allowing threads to communicate with one another directly to complete their operations *without* modifying the stack. We describe a technique for dictionary operations called *coalescing* in which multiple concurrent insertions and deletions of the *same key* are grouped and reordered (i.e., their linearization order is permuted) such that they can all be completed with at most *one* modification to the data structure.

In particular, after a simple insert (i.e., one that does not split a leaf), or a successful delete, we try to identify and coalesce any concurrent insertions and deletions of the same key[1]. Suppose we linearize this successful update operation U as in Chapter 4.4. Then, linearizability allows us to freely order other insertions and deletions of `key` before or after U, as long as they are concurrent with the linearization point of U.

Suppose U is a simple `insert(key, val)`. If a deletion of `key` is concurrent with the linearization point of U, then it can be linearized immediately before U and return $\perp$ (without modifying the data structure). Similarly, if an insertion of `key` is concurrent with the linearization point of U, then it can be linearized immediately after U and return

[1]We could potentially also coalesce after performing a splitting insert, but splits are relatively uncommon, and this would complicate the algorithm and its correctness argument.

`val`. Since neither of these operations change the data structure, an arbitrary number of insertions and deletions of `key` can be coalesced in this way, provided they are concurrent with the linearization point of U.

The case where U is a successful `delete(key)` is similar. A deletion of `key` that is concurrent with the linearization point of U is linearized after U (and returns $\perp$), and an insertion of `key` that is concurrent with the linearization point of U is linearized before U (and returns the value removed by U).

5.1 Coalescing with the MCS queue

The challenge is now to detect insertions and deletions of `key` that are concurrent with the linearization point of a simple insert or successful delete. Since MCS locks were already being used to improve scaling, an interesting initial approach is to leverage the queue to do this: the thread that performed the successful operation traverses the queue, coalescing any operations on the same key (and removing their queue nodes from the queue). This approach is somewhat similar to flat combining [29], but with the key difference that a coalesced thread can return immediately after it is coalesced instead of having to wait for coalescing to complete. The resulting tree is called the Coalescing-MCS-ABtree, or CoMCS-ABtree.

Algorithm 5.1: MCS coalescing implementation

```
abstract type Node
  queueTail : MCSNode
  ... // remaining fields

type Operation is INS, DEL or BAL

type MCSNode
  op          : Operation
  key         : K
  next        : MCSNode
  retval      : V
  owned       : bool
  coalesced   : bool

<bool, V> lockOrCoalesce(node, mcsNode)
  // atomically set tail
  tail = atomic_swap(node.queueTail, mcsNode)
  if tail ≠ NULL
    tail.next = mcsNode
    Spin until owned or coalesced is set
    if mcsNode.coalesced
      return <false, retval>
  return <true, ⊥>

// Same as regular MCS lock
release(mcsNode, node)
  if CAS(node.queueTail, mcsNode, NULL) succeeds
    return
  // Another node is in queue
  Spin until next is set
  next.owned = true

coalesce(head, end, val)
  if end = head return

  Spin until head.next is set
  prevInQueue = head
  curr = head.next
  while curr ≠ end
    Spin until curr.next is set
    nextCurr = curr.next

    if curr.key ≠ head.key or curr.op = BAL
      // Do not coalesce curr
      prevInQueue.next = curr
      prevInQueue = curr
    else
      // Coalesce curr
      curr.retval = val
      curr.coalesced = true
    curr = nextCurr
  prevInQueue.next = end

V insert(key, val)
  myNode = new MCSNode(key, INS)
  acq,retval = lockOrCoalesce(leaf, myNode)
  if not acq
    return retval
  ...
  if leaf.keys[i] = ⊥
    leaf.ver++
    ... // Insert key
    currTail = leaf.queueTail
    leaf.ver++
    coalesce(myNode, currTail, val)
    Unlock leaf
    return ⊥
```

Algorithm 5.1 contains the pseudocode for the CoMCS-ABtree. Each node contains a queue of *augmented* MCS queue nodes. In addition to the standard `owned` and `next` fields, an augmented queue node created by an update U contains the following data:

- `op`: The type of operation (insert, delete, or rebalance) U wishes to perform
- `key`: `key` if U is an `insert(key, val)` or `delete(key)`, otherwise $\perp$
- `coalesced`: A bit that indicates the insert/delete has been coalesced (initially false)
- `retval`: The value to return if U is an `insert` and is coalesced

To lock a node, a thread first creates an augmented MCS queue node then invokes `lockOrCoalesce`, which returns when either the lock is acquired or the thread's operation is coalesced. `lockOrCoalesce` appends the queue node to the end of the queue by atomically swapping the node's existing tail with the new queue node then making the existing tail's next point to the queue node. The thread then spins not only on its `owned` variable (as in a traditional MCS lock), but also on its `coalesced` bit. Thus, a thread will stop spinning either because it holds the lock, or because it has been coalesced. If it was coalesced, the thread immediately returns `retval` if it was performing an insert and $\perp$ if it was performing a delete. (How `retval` is set is described below. Recall from the discussion in the beginning of this chapter that all coalesced deletes return $\perp$.) A thread performing a balance operation is never coalesced. If the lock is acquired, the thread continues to perform its operation.

Every simple insert and successful delete will attempt to perform coalescing. Let U be a simple insert or successful delete, l be the leaf it modifies, and v the value that U deletes or inserts. *After* U finishes modifying l, but *before* its second write to the version of l, U reads the current MCS queue tail t. U then increments the version and invokes `coalesce`, which performs a bounded traversal starting from the head of the queue and continuing until it reaches t (`coalesce` is described further below). Whenever `coalesce` encounters a queue node that represents an insertion or deletion of the same `key`, it *removes* the node from the queue, stores v in `retval`, and finally sets `coalesced` to `true`. As discussed in the beginning of this chapter, v is the correct value for coalesced inserts to return.

Note that U reads the queue tail *before* it is linearized (i.e., before it performs the second write to the version of l). Thus, every node U encounters during `coalesce` is in the queue when U is linearized. In other words, every removed node represents an operation that is concurrent with the linearization point of U. It is crucial that the queue tail is read before the linearization point of U. Otherwise, U could traverse nodes belonging to operations that started *after* U was linearized, and erroneously coalesce them (which is not correct, because such operations cannot necessarily be linearized as described in the beginning of this chapter).

Note that `coalesce` is only invoked on leaf nodes, so the MCS queues in internal nodes are just regular MCS queues (without any coalescing). Coalescing greatly improves performance at high thread counts in high contention workloads, as Figure 5.1 shows.

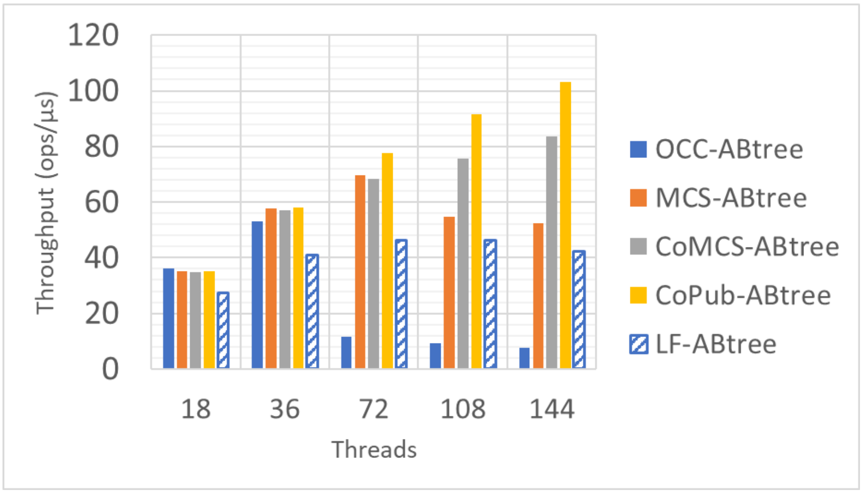

Figure 5.1: (a,b)-tree comparison on high-contention workload.

5.1.1 `coalesce` explanation

`coalesce` removes all nodes that represent insert or delete operations on `head.key` between `head` and `end`, the tail read at line 262 (excluding `head` and `end`). Note that `head` is indeed the head of the queue since `coalesce` is only invoked if the lock is owned and the owning thread's `MCSNode` is always the head of the queue.

If `end = head`, there are no nodes strictly between `head` and `end` so `coalesce` returns without doing anything. Otherwise, `coalesce` begins reading the MCS queue starting from the node after `head` (which must exist since `end` is somewhere after `head` in the queue). `prevInQueue` stores the last traversed queue node which was not coalesced, and is initially `head`.

In the loop, the thread waits for the current node's successor to be set and saves it in a local variable before deciding to whether to coalesce the current node. This is necessary in this implementation because MCS queue nodes are *stack-allocated*. This means that once a queue node is coalesced, its owning thread might return and overwrite the queue node's data (including the pointer to its successor in the queue). Thus, the successor must be saved before marking the node as coalesced.

If the current node's key does not match `head.key` or if the queue node's operation is not `INS` or `DEL` (i.e. it is a balance operation, `BAL`), the queue node is not coalesced. So, the last not coalesced queue node (`prevInQueue`) is linked to `curr`. If the node should be coalesced, its return value is set to `val` and its `coalesced` bit is set. `prevInQueue` does not

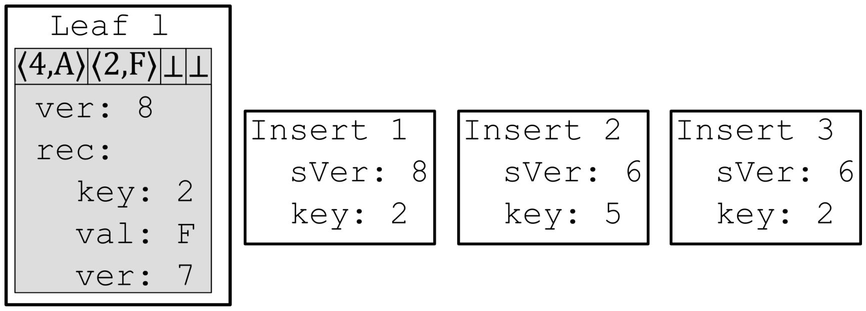

Figure 5.2: Coalescing publishing example. Consider the state of leaf l as shown. $l.rec$ stores the `CoRec` of a completed simple insert `insert(2,F)`. Consider three (independent) inserts that are attempting to insert in l and are all at line 286. Insert 1 cannot coalesce itself with `rec` since the version of the leaf it read is greater than `rec.ver`. Insert 2 cannot coalesce itself since its key does not match `rec.key`. Insert 3 can coalesce itself.

change in this case. After the loop is complete, `prevInQueue` is linked to `end`, connecting the non-coalesced nodes to the remainder of the queue (from `end` onwards).

`coalesce` is guaranteed to reach `end` eventually, since queue nodes cannot leave the queue unless they acquire the lock or are coalesced by `coalesce`. The thread that invokes `coalesce` holds the lock for the entire duration of `coalesce` and no other thread can invoke `coalesce` during this time since `coalesce` is only invoked by a thread that holds the lock.

5.2 Coalescing publishing

In the CoMCS-ABtree, queue traversals cause costly cache misses. This section describes a modification of the OCC-ABtree called the coalescing publishing (a,b)-tree (CoPub-ABtree, for short), in which each leaf stores a summary, called an `CoRec`, of the last operation U that *modified* it, which concurrent operations use to coalesce *themselves*. (U is either a simple `insert` or a successful `delete`.)

The contents of an `CoRec` are similar to the data that MCS queue nodes are augmented with in Section 5.1: `key` stores the key that U inserted or deleted and `val` stores the value that it inserted or deleted. Thus, if an insert on the same key as U can determine that it is concurrent with the linearization point of U, it can coalesce itself and return the `val` from the `CoRec` published by U. Similarly, if a delete on the same key as U is concurrent with the linearization point of U, it can coalesce itself and return $\perp$.

The `ver` field of `CoRec` is used by inserts and deletes to determine whether they are concurrent with the linearization point of U. Let l be the leaf modified by U. U increments the version of l to an odd value v, performs its modification, publishes an `CoRec` in l whose `ver` field set to v, and finally increments the version of l to $v + 1$. Recall that the linearization point of U is when it sets the version to $v + 1$ (i.e., the second increment of the leaf's version). Thus, an insert/delete is concurrent with the linearization point of U if:

1. It reads a version of l less than or equal to v **and**
2. It returns after the version of l is at least $v + 1$

since the version number of a leaf only ever increases. Condition 1 guarantees that the insert/delete is present before the linearization point of U, and condition 2 guarantees that it is present after the linearization point.

Let us see how an `insert(key, val)` decides whether it can coalesce itself (Algorithm 5.2). As soon as the insert's search terminates, it reads the leaf's version and stores it in `sVer` (short for *search version*). As in the OCC-ABtree, it then traverses the leaf and if it finds `key`, it returns the associated value. If it does not find `key`, or if the leaf is being modified (i.e., if its version is odd), the insert invokes `lockOrCoalesce` in an attempt to coalesce itself, or lock the leaf if it is unable to do so.

In `lockOrCoalesce`, the insert attempts to read a snapshot of the leaf's `CoRec`. To do this, it reads the leaf's version (line 280), then reads the `CoRec rec`, then re-reads the leaf's version (line 284). If the reads of the leaf's version return identical results, and the version is even (indicating the leaf is not being modified), then a snapshot was obtained. Otherwise, `lockOrCoalesce` tries to obtain a snapshot again.

Once a snapshot is obtained, condition 2 is guaranteed to be satisfied. To see why, note that the leaf's version is even when it is last read at line 283 by the exit condition of the loop. But, `rec.ver` is always an odd value, thus the version read at line 283 is at least `rec.ver+1`.

At line 286, `lockOrCoalesce` tries to determine whether condition 1 is satisfied. If it is, and `key` matches `rec.key`, then `lockOrCoalesce` returns `<false, rec.val>`, so `tryInsert` returns at line 321, which causes the insert to coalesce itself and return `rec.val`. Otherwise, the insert does not have enough information to argue that it can coalesce itself, so it attempts to lock the leaf. If it acquires the lock, the insert proceeds as in the OCC-ABtree. If it performs a simple insert, it publishes a `CoRec` (as described above). If the insert fails to acquire the lock, it attempts to coalesce itself again.

The coalescing of deletes is similar, except that coalesced deletes always return $\perp$ (not `rec.val`). Figure 5.2 shows an example of coalescing publishing.

Without the need to traverse a queue, the CoPub-ABtree scales even better than the CoMCS-ABtree, which already outperforms the OCC-ABtree and MCS-ABtree on the high-contention workload (Figure 5.1).

Finally, we note that the `CoRec` could also be used to linearize `find`s in high-contention workloads. In some extreme scenarios, this could possibly be useful in preventing `find(key)` from being starved by an endless stream of updates to `key`. I did not observe this in my experiments, since the nodes are small enough that searches can typically traverse a leaf in the interval between when one update completes and the next one begins.

Algorithm 5.2: Coalescing publishing

```
// K is key type, V is value type
type CoRec
  key : K
  val : V
  ver : int

type Leaf
  rec: CoRec
  ... // remaining fields

<bool, V> lockOrCoalesce(leaf, key,
   sVer)
  while true
    // Try to coalesce self
    do
      ver1 = leaf.ver
      rec = leaf.rec
      ver2 = leaf.ver
    while ver1 is odd or ver1 ≠ ver2

    if sVer ≤ rec.ver and rec.key =
       key
      return <false, rec.val>

    // Cannot coalesce, try to lock
    if tryLock(leaf.lock)
      return <true, _>

<RetCode, V> trySearchLeaf(leaf, key)
  ver1 = leaf.ver
  if ver1 is odd
    return <RETRY, ⊥>

  val = ⊥
  for i = 0 up to MAX_SIZE - 1
    if leaf.keys[i] = key
      val = leaf.vals[i]
      break
  ver2 = leaf.ver
  if ver1 ≠ ver2 return <RETRY, ⊥>
  if val = ⊥ return <FAILURE, ⊥>
  else return <SUCCESS, val>

<RetCode, V, PathInfo, int>
    trySearch(key)
  Search for leaf as before
  sVer = leaf.ver
  rc, val, ver = trySearchLeaf(
      leaf, key)
  return <rc, val, path, sVer>

V insert(key, val)
  rc, prev, path, sVer =
      trySearch(key)
  if rc = SUCCESS return prev

  leaf = path.n
  parent = path.p
  acq, retval = lockOrCoalesce(
      leaf, key, sVer)
  if not acq
    return retval

  // Did not coalesce, insert as
       usual
  leaf.ver++
  ... // Insert key
  leaf.rec = <key,val,leaf.ver>
  leaf.ver++
  Unlock leaf and return ⊥
  ...
```

Chapter 6

Experiments

This chapter compares the trees presented thus far with other leading dictionary implementations using the SetBench benchmarking framework.

6.1 Setup and methodology

6.1.1 System

The experiments in this chapter were run on a 4-socket Intel Xeon Gold 5220 with 18 cores per socket and 2 hyperthreads (HTs) per core, for a total of 144 hardware threads, and 192GiB of RAM. In all experiments, threads are pinned such that the first socket is saturated before the second socket is used, and so on. Additionally, the pinning ensures that all cores on a socket are used before hyperthreading was engaged. The machine runs Ubuntu 20.04.2 LTS. All code is written in C++ and compiled with G++ 7.5.0-3 with compilation options `-std=c++14 -O3`. The scalable allocator jemalloc 5.0.1-25 is used. `numactl -i all` is used to interleave pages evenly across all processor sockets.

6.1.2 Memory reclamation

All data structures in the experiments use a fast variant of epoch based memory reclamation called DEBRA [14], except the SplayList (which does not reclaim memory at all) and the OpenBw-Tree (which uses a different form of EBR and was too complex to port to DEBRA).

6.1.3 Methodology

Each experiment *run* starts with a prefilling phase, in which a random subset of keys are inserted into the data structure until the data structure size reaches its expected size in the steady state (half of the key range, in these experiments, since the proportions of inserts and deletes are always equal). After the prefilling phase, n threads are created and started together, and the *measured* phase of the experiment begins. In this phase, each thread repeatedly selects an operation (`insert`, `delete`, `find`) based on the desired update frequency, selects a key according to a uniform or Zipfian distribution. This continues for 10 seconds, and the total *throughput* (operations completed per second) is recorded. Each experiment is run three times, and the results below are the averages of the runs.

6.1.4 Validation

To sanity-check the correctness of the evaluated data structures, each thread keeps track of the sum of keys that it inserts and deletes into the tree. At the end of each run, all threads' sums are added to a grand total, and the grand total must match the sum of keys in the data structure.

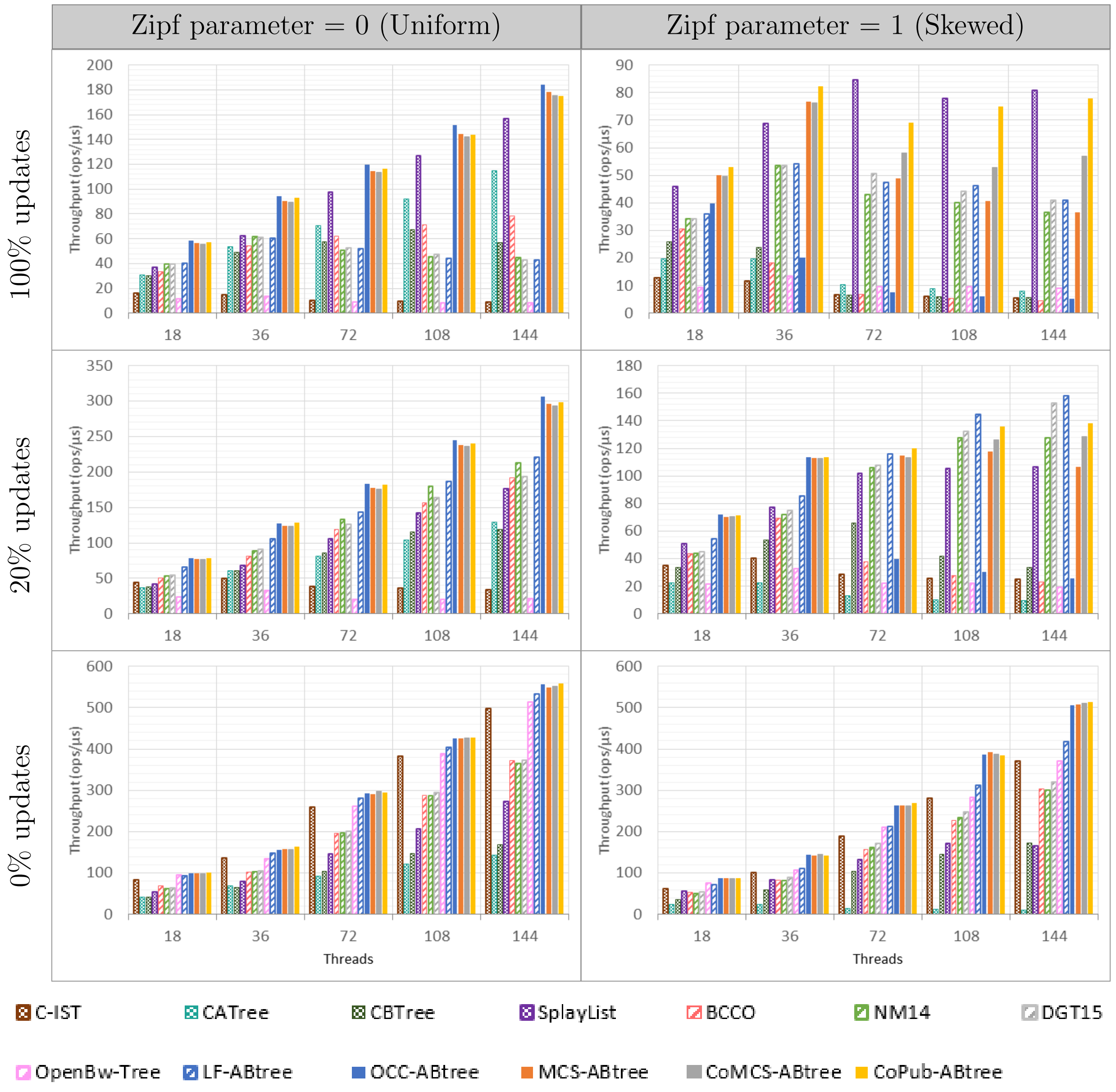

Figure 6.1: Results of benchmark with **10000 keys**.

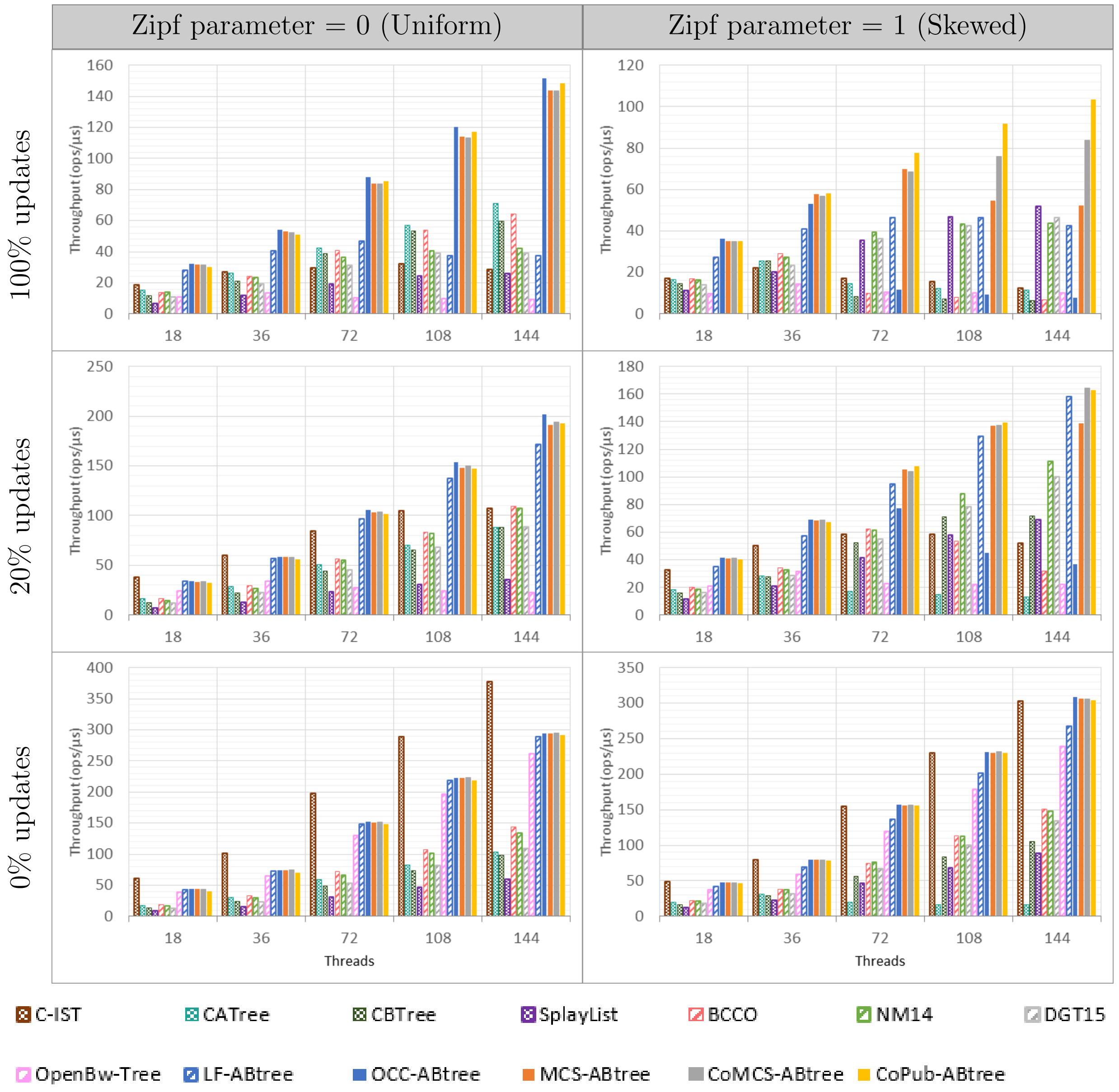

Figure 6.2: Benchmark with **1M keys**.

6.2 Results discussion

See Chapter 3 for descriptions of the data structures included in the experiment graphs. In Figures 6.1 and 6.2, solid bars represent the trees from this thesis, striped bars represent data structures that are distribution-naïve (LF-ABtree, BCCO10, NM14, DGT15, OpenBw-Tree), and checkered bars represent data structures that adapt their structure to the access distribution (CATree, CBTree, SplayList), or try to exploit it to obtain faster searches (the concurrent interpolation search tree, C-IST in the figure).

6.2.1 Effect of update frequency and distribution

Search-only

In general, better performance on the search-only workloads is correlated with short paths to the keys, since longer paths result in more cache misses. The C-IST has nodes that are much larger than those of the (a,b)-trees and the OpenBw-Tree, and quickly traverses them using interpolation search. For example, the root in the C-IST contains $O(\sqrt{n})$ keys, where n is the size of the entire tree. As a result, the C-IST is very shallow and has extremely fast searches. The (a,b)-trees and the OpenBw-Tree have heights larger than the C-IST but less than the BSTs. Accordingly, in the 1 million key workload, the (a,b)-trees and OpenBw-Tree perform worse than the C-IST and much better than the BSTs (BCCO10, NM14, DGT15), roughly doubling their throughput. On the 10000 key workload, the C-IST performs slightly worse than the (a,b)-trees, since it is optimized for large data structures. The BSTs have similar performance relative to one another on the search-only uniform workloads.

The CBTree and SplayList performed worse than expected on the search-only, Zipfian workload. Splaying in these data structures should theoretically accelerate searches (especially since the splayed key is never removed in a search-only workload), but they barely exceed their performance on the uniform workload. The CATree's performance is reasonable on the uniform workload, but is much worse than the other data structures on the Zipfian workload. This observation is true in the workloads with updates, as well, since all of the CATree's operations (even searches) require locking the leaf that is being searched or modified. The CATree's authors describe a lock-free CATree, but we could not find a high performance C/C++ implementation, and according to their own results, their optimized implementation does not scale beyond 16 threads. The poor performance of the CBTree, SplayList, and CATree illustrates a common flaw of distribution-adaptive data structures: they rely on arbitrary parameters that determine how often adjustments should be performed, and these parameters may not be optimal for all systems (or thread counts). The parameters used in these experiments are the default parameters in the code provided by the authors. There is no obvious way to improve the parameters without specializing the results to target a certain update frequency, key range, and key access distribution.

The distribution-naive data structures exhibit the similar performance on the search-only uniform and Zipfian distributions.

The implementation of the CATree used in these experiments is a port of the authors' Java code. The authors also describe a lock-free CATree and an optimization to the CATree that makes it more performant in high-contention settings by making some leaf nodes lock-free, but I was unable to port these versions of their Java code to C++ because of their complexity. However, the performance of the locking CATree implementation in these experiments seems to match the performance in the original paper [51]. Per their own results, the optimized version does not scale past 16 threads, and even assuming the same (approximately 2x) performance benefit they observe from the optimization, the lock-free optimized CATree would still likely be slower than the trees presented in this thesis.

20% and 100% updates

The CoMCS-ABtree and CoPub-ABtree drastically outperform the other data structures in the 100% Zipfian update workload, with the CoPub-ABtree nearly doubling the performance of its closest competitors.

Data structures that perform much worse in the update workloads can be divided into two groups. The first group is comprised of data structures that have slow updates even when keys are uniformly distributed. This group includes the LF-ABtree, OpenBw-Tree, and the C-IST, all of which perform much worse on 100% uniform updates than in the search-only workload, especially when running on multiple processor sockets (72 threads and above). The LF-ABtree handles a moderate amount of updates well (outperforming all non-coalescing data structures at 144 threads), but the OpenBw-Tree and C-IST do not. In the case of the C-IST, this is because rebuilding the tree after updates is extremely costly. The second group is comprised of data structures whose updates do not scale well with contention (i.e., in the Zipfian workload). This group includes the OCC-ABtree (for the reasons mentioned in Chapter 4), the CATree (for the reasons mentioned above), and BCCO10, which uses OCC and might suffer from the same problems that affect the OCC-ABtree.

A notable outlier in these results is the SplayList, which had relatively poor search-only performance but exceeds the performance of NM14, DGT15, and the LF-ABtree on the high-contention workload. This may be partially because the SplayList never removes deleted keys from the data structure, so reinserting a key that was once in the SplayList requires no memory allocation (allocation normally adds considerable overhead to the other data structures). This approach is quite efficient for the purposes of this benchmark, but might be less so if the set of keys that are *ever* inserted is much larger than the set of keys that are *typically* in the dictionary. In that case, in addition to being slower, the SplayList would also waste memory.

6.2.2 Effect of key range

Overall, most data structures perform worse on the larger key range workload. This is because path lengths to a leaf are (on average) longer when there are more keys in the data structure. As discussed in the previous section, longer path lengths result in more cache misses hence slower operations.

However, there is an important exception to this rule. For update-heavy workloads, the trees whose nodes have multiple nodes ((a,b)-trees, the C-IST, and the Open Bw-Tree) perform no worse or even *better* on larger key ranges. In the larger key range workload, two *random* keys are less likely to share a leaf than in the smaller key range workload (regardless of whether the accesses are uniform or Zipfian). This results in less contention on locks, because operations on two random keys are less likely to acquire the same lock. In addition, for the CoMCS-ABtree and CoPub-ABtree in Zipfian workloads, coalescing is more likely to occur in the larger key range workload because most operations accessing a node are acting on the same key.

The SplayList performs even worse than the other data structures on the larger key range. This is likely because operations must search through every key that was ever in the data structure, even if it has been deleted (since the SplayList does not remove deleted keys from the data structure).

Chapter 7

Persistent (a,b)-trees

This chapter describes the changes required to make the OCC-ABtree strictly-linearizable in a system with persistent main memory. The resulting tree, the Persistent Memory OCC-ABtree (or p-OCC-ABtree for short), requires significantly fewer cache line flushes than one of the best-performing persistent B-trees, the FPtree [46]. The MCS-ABtree, CoMCS-ABtree, and CoPub-ABtree can also be modified in the same way to yield strictly-linearizable implementations. The end of the chapter provides experimental results comparing the persistent memory trees to their volatile memory counterparts, and shows that the persistent memory trees are only slightly slower.

7.1 The p-OCC-ABtree

As the name suggests, the p-OCC-ABtree is based on the OCC-ABtree. The defining difference between the two algorithms is that the p-OCC-ABtree operations guarantee that their changes to keys, values, and pointers are in *persistent* memory before they return. This allows the p-OCC-ABtree to be recovered to a consistent state after a crash. For performance reasons, the remaining node fields (e.g. size, version, lock state) are not persisted. Thus, they might contain incorrect values after a crash. However, they can easily be fixed by the recovery procedure.

Many concepts from the OCC-ABtree have a direct analogue in the p-OCC-ABtree. For example, the concept of reachability in the OCC-ABtree is analogous to p-reachability in the p-OCC-ABtree. A node is **p-reachable** (short for persistently reachable) if it can be reached from the entry node by following child pointers in *persistent* memory. The p-OCC-ABtree also satisfies many of the same invariants as the OCC-ABtree. For example, updates to the p-OCC-ABtree appear to take effect atomically. In fact, all the invariants proved for the OCC-ABtree have an analogous version in the p-OCC-ABtree (described in the correctness section).

The implementation of the p-OCC-ABtree is similar to the implementation of the OCC-ABtree. However, there are some changes that must be made to guarantee its correctness in the persistent memory model. These are described in the next section.

Figure 7.1: Selected p-OCC-ABtree operations

```
V insert(key, val)
  ...
  if leaf.size < MAX_SIZE
    // Insert without splitting
    for i = 0 to MAX_SIZE - 1
      if leaf.keys[i] = ⊥
        leaf.ver++
        leaf.vals[i] = val
        flush(leaf.vals[i])
        sfence()
        leaf.keys[i] = key
        flush(leaf.keys[i])
        sfence()
        leaf.size++
        leaf.ver++
        Unlock leaf and return ⊥
  else
    Lock parent
    if parent marked
      Unlock leaf/parent
      goto RETRY

    // n is tagged if parent is not entry
    Create internal n with two new children c1 and c2 that evenly share contents of leaf and new key/val
    flush(c1)
    flush(c2)
    flush(n)
    sfence()
    parent.ptrs[path.nIndex] = marked ptr to n
    flush(parent.ptrs[path.nIndex])
    sfence()
    unmark(ptr)
    Mark leaf
    Unlock leaf/parent
    cleanup(n.searchKey)
    return ⊥
```

```
<RetCode, V, PathInfo> search(key)
  gp = NULL, p = NULL
  pIndex = 0, nIndex = 0
  n = entry
  do
    gp = p, p = n, pIndex = nIndex
    nIndex = seqGetChild(n, key)
    Spin while n.ptrs[nIndex] is marked
    n = n.ptrs[nIndex]
  while n is not leaf

  path = PathInfo(gp, p, pIndex, n, nIndex)
  rc, val = searchLeaf(n, key)
  return <rc, val, path>



recover(node)
  Unlock node if locked
  Unmark node if marked
  if node is Leaf
    size = number of non-⊥ keys in node
    node.ver = 0
    flush(node)
  else
    for each ptr in node.ptrs
      if ptr marked
        unmark(ptr)
      recover(child at ptr)

recoverTree()
  recover(entry)
  sfence()
```

7.2 Implementation

The goal of this section is to provide the reader with an intuitive understanding of the differences between the OCC-ABtree implementation and the p-OCC-ABtree implementation. More rigorous arguments appear in the correctness section.

The implementation of the p-OCC-ABtree differs from the OCC-ABtree in three ways. First, the p-OCC-ABtree uses the link-and-persist method from [20] to link in new updates. At a high-level, this ensures that updates only modify p-reachable nodes, and searches only visit nodes that were p-reachable at some time during the search. Second, updates use `flush` and `sfence` to guarantee that they appear to take effect *atomically* in the p-OCC-ABtree. Finally, the p-OCC-ABtree includes a recovery procedure, `recoverTree`, which is invoked after a crash to bring the tree to a valid state.

Algorithm 7.1 contains the pseudocode for selected operations in the p-OCC-ABtree. `flush(x)` is used as shorthand for flushing all cache lines that contain object `x`.

7.2.1 Link-and-persist

To motivate the link-and-persist method, consider the following scenario. Suppose a splitting insert inserts `key` as it would in the OCC-ABtree (by splitting a leaf node, creating a tagged node joining the two new leaves, and changing a pointer in the tree to point to the tagged node). Recall that the pointer to the tagged node is first written into volatile memory (caches) and later flushed to persistent memory (main memory). Suppose that a crash occurs *after* the pointer to the tagged node is written to volatile memory, but *before* the pointer is flushed to persistent memory (i.e. before the tagged node or its children become p-reachable). Since the pointer is not persisted, the insert *cannot* be linearized. Not linearizing the insert is allowed by strict linearizability; operations that are interrupted by a crash can be *dropped* (i.e. removed from the execution).

However, a `find(key)` that took place after the pointer was written into volatile memory might have found `key`, returned its associated value `val`, and terminated before the crash. This `find` cannot be linearized because the `insert` that inserted `key` and `val` was removed from the execution. Unlike the `insert`, `find` was *not* interrupted by the crash, and so strict-linearizability does not allow us to remove it from the execution (so the execution is not strictly-linearizable).

The problem above occurred because `find` returned data that was not yet p-reachable (i.e., was only reachable in volatile memory). This can be fixed by using the link-and-persist technique [20] (a similar technique is given in [56]). Algorithm 7.1 shows the modifications to splitting inserts and `search`. (Rebalancing steps are modified similarly to splitting inserts, and `cleanup` is modified similarly to `search`.)

In the link-and-persist method, an update's new nodes are linked into the tree by changing a pointer to point to them (like in the OCC-ABtree). However, the pointer is initially created with a mark on it (line 358), to indicate that the pointer might not be persisted (and thus the node(s) it points to might not be p-reachable). The update then flushes the marked pointer and performs an `sfence` (lines 359-360). Finally, after the `sfence`, the update removes the mark from the pointer (361).

The searches in `search` and `cleanup` are modified to wait until a pointer is not marked before following it (line 373). This ensures that `search` and `cleanup` only access nodes that have been p-reachable. All operations use `search` or `cleanup` to find any nodes they will modify. Moreover, as shown in the correctness section, updates only modify nodes that are *currently* p-reachable.

7.2.2 Atomic updates with `flush` and `sfence`

A recovery procedure must be able to recover a consistent state of the data structure after a crash. If it is possible for the persistent data structure to contain partial modifications from an operation, the recovery procedure must be able to either complete or rollback the operation. A common way to achieve this is to *log* the changes an operation will perform before performing those changes. If a crash occurs, the recovery procedure can use the log to complete or revert any incomplete operations. Logging can incur significant overhead because the log itself must be flushed to persistent memory.

I take a simpler approach: for each update, there is a single flush to persistent memory when the update appears to take place.

Simple insert. Instead of just writing `val` and `key`, a simple insert now does the following (see lines 333 to 345):

1. `val` is written into the node
2. `flush` is performed on the cache line c_v containing `val`
3. An `sfence` is performed
4. `key` is written into the node
5. `flush` is performed on the cache line c_k containing `key`
6. Another `sfence` is performed

The inserted key-value pair appears atomically (in the appropriate leaf, which is already p-reachable) when c_k is flushed to persistent memory. Note that a background flush might cause this to occur *before* the explicit `flush` at step 5 (but after step 4). Before c_k is flushed, the key slot contains $\perp$, so the key-value pair is logically not present in the tree (since $\perp$ is not a valid key).

Successful delete. In a successful `delete(key)`, `key` is set to $\perp$, a `flush` is performed on the cache line (containing `key`), followed by an `sfence`. The delete takes effect atomically when $\perp$ is flushed (since $\perp$ is not a valid key).

Other updates. All other updates to the tree (splitting inserts and rebalancing steps) behave as follows (see lines 347 to 365):

1. The new nodes are created
2. `flush` is performed on all cache lines that contain new nodes
3. An `sfence` is performed
4. The new nodes are linked into the tree by changing one pointer
5. `flush` is performed on this pointer
6. Another `sfence` is performed

Flushing the new nodes before linking them into the tree *prepares* the update in persistent memory (without actually introducing it to the tree). The update then takes effect atomically when the pointer to the new nodes is flushed (either explicitly, or by a background flush between steps 4 and 6).

7.2.3 Recovery

Because updates to the p-OCC-ABtree are atomic, the recovery procedure does not have to deal with partial modifications to the tree. The recovery procedure is `recoverTree`. `recoverTree` invokes `recover` on the entry node (which is assumed to be at a known location after a crash). `recover` fixes a node as described below and then recursively invokes itself on all children of its node.

The recovery procedure makes one change to guarantee the safety of the tree: it fixes the size field to match the actual number of keys in a leaf, in case the persisted value does not match the actual value. The other four changes made by the recovery procedure restore the deadlock-freedom guarantee of the p-OCC-ABtree. Overall, the recovery procedure does the following. At each node:

- The lock state is set to unlocked.
- The marked bit is unset.
- If the node is a leaf, its version is reset to 0.
- If the node is a leaf, its size field is set to the number of keys in the leaf.
- If the node is an internal node, any marked child pointer is unmarked. The pointer is already persisted since it is being accessed from persistent memory.

Any changes made during recovery are flushed and an `sfence` is performed at the end of recovery. The recovery procedure does not need to repair a node's `searchKey` or the `size` field of internal nodes, since these fields are constant from the time they are initialized.

7.3 Correctness

This section begins by providing a definition to link the physical state of the p-OCC-ABtree to its abstract contents. It then mentions some invariants which hold for the p-OCC-ABtree; these are analogous to the invariants of the OCC-ABtree and can similarly be used to show that the p-OCC-ABtree is strictly-linearizable.

7.3.1 Definitions

The system is said to be **recovering** from the time when a crash occurs until the time when `recoverTree` returns.

In strict linearizability, every operation that is concurrent with a crash must either be linearized before the crash or be removed from the execution. Any simple insert or successful delete that has flushed a `key` will be recovered (and thus cannot be removed from the execution). These operations must therefore be linearized before the crash, even if they have not yet incremented the version for the second time. This is reflected in the definition below, and in the changes to the linearization points in the following section.

Definition 12. Let l be a p-reachable node. A key k (not equal to $\bot$) is **in the p-OCC-ABtree** if either

1. The system is recovering and k is in l's keys array OR
2. The system is not recovering and k was in l's keys array when l's version (in *volatile* memory) was last even

Furthermore, if key k is the ith key in l, the value associated with k is `l.vals[i]`.

If the system is not recovering, Definition 12 is similar to the definition of a key being in the OCC-ABtree. That is, keys and values are logically added or removed from the tree when the version number is incremented to an even number. If the system is recovering, however, every key in a p-reachable node is in the tree (the version is ignored).

7.3.2 Invariants

Theorem 13 (p-OCC-ABtree Invariants)**.** *The p-OCC-ABtree satisfies the following invariants, which are analogous to the OCC-ABtree invariants:*

1. *The p-reachable nodes form a relaxed (a,b)-tree.*
2. *The key range of a node that was once p-reachable is constant.*
3. *A node that is not p-reachable contains the same keys and values that it did when it was last p-reachable and unlocked (i.e. updates do not both unlink and modify a node).*
4. *A key appears at most once in a leaf.*
5. *If a node was once p-reachable, and is currently unmarked, it is p-reachable.*
6. *If a node is unlocked and was once reachable, its `size` field matches the number of keys it contains.*
7. *The key range of `n` in `search(key)` contains `key`.*

Proof. The proofs of most of these invariants are similar to the proofs in Section 4.4. The proof for invariant 1 requires an additional explanation of why every node used by an update was once p-reachable.

Proof of invariant 1. Recall that to prove the p-OCC-ABtree is a relaxed (a,b)-tree, it suffices show that for each update:

- There is a single step at which the update appears to take place
- The update is correct

The first condition holds for the reasons laid out in the previous section on atomic updates.

The second condition is largely the same as the proof in the OCC-ABtree. However, that proof uses invariant 5, which requires showing that the nodes traversed in the search were all reachable at some time. This was trivial in the case of the OCC-ABtree (because the nodes are reached by following child pointers), but is not trivial in the p-OCC-ABtree, which uses *p-reachability*.

We will show that every node traversed by `search` in the p-OCC-ABtree was p-reachable at some time. Assume that every node traversed by a search until node n is p-reachable. If n is the entry node, it is p-reachable by definition.

Otherwise, the search reached n by following an unmarked pointer from a node p. We will show that there exists a time t when p was p-reachable and contained the unmarked pointer to n. If p was p-reachable when it read the unmarked pointer to n, t is the time of the read. Otherwise, invariant 3 guarantees that p's pointers have not been modified since it was last p-reachable. Thus, when p was last p-reachable, it contained an unmarked pointer to n. In this case, t is the time when p was last reachable.

Finally, notice that there are two ways p could contain an unmarked pointer to n. The first way is if p contained a pointer to n when it was created. In this case, since p was flushed before being linked into the tree, its pointer to n is persisted. Otherwise, the pointer was first introduced to p by an update U as a marked pointer. This update must have flushed the pointer before unmarking it. In either case, the unmarked pointer was in persistent memory by time t.

At time t, p was p-reachable and contained a pointer to n in persistent memory. Thus, n was p-reachable at time t.

The remainder of the proof of is similar to the proof for the OCC-ABtree. □

7.3.3 Strict linearizability

The p-OCC-ABtree has slightly different linearization points than the OCC-ABtree, to deal with the different definition of when a key is in the tree.

Operations which do not modify the tree are linearized as in the OCC-ABtree. Splitting inserts in the p-OCC-ABtree are linearized when the pointer to the new nodes is flushed to persistent memory (instead of it is written to volatile memory).

Simple inserts and successful deletes linearize differently depending on whether or not they are interrupted by a crash. When not interrupted by a crash, simple inserts and successful deletes have the same linearization points as they did in the OCC-ABtree: the increment of the leaf's version (in volatile memory) to an even number. Recall that this linearization point is chosen to support coalescing publishing.

To see why we cannot linearize the same way when interrupted by a crash, consider the following scenario. Suppose a simple insert or successful delete that flushes `key` (thus making its change persistent) but does increment the version to an even number before a crash. The recovery procedure would recover this key-value pair, even though the operation was not linearized. To solve this problem, we linearize these operations *at the crash.*

Theorem 14. *The p-OCC-ABtree is strictly-linearizable.*

Proof. Note that these linearization points all occur after an operation's invocation and before its response or crash. We must show that performing each operation at its linearization point (and returning the appropriate value) correctly affects the contents of the abstract dictionary according to Definition 12. Let E be an arbitrary execution of the p-OCC-ABtree. We prove that E is strictly-linearizable by induction.

Suppose the prefix of E up to the beginning of the ith era of operations (including the recovery procedure after the $i-1$th crash, if $i > 1$) is strictly-linearizable, and that the p-OCC-ABtree satisfies all invariants. We show that the prefix up to the beginning of the $i+1$th era of operations is strictly-linearizable and the p-OCC-ABtree recovered after the ith crash satisfies all invariants.

We do this by breaking up the execution fragment from the beginning of the ith era of operations to the beginning of the $i+1$th era of operations into three parts: the execution fragment before the ith crash, the ith crash, and the recovery after the ith crash. We show that each fragment is strictly-linearizable by showing that the operations correctly modify the abstract dictionary. Note that the concatenation of strictly-linearizable execution fragments is strictly-linearizable, by the *locality* property of strict linearizability.

Before the ith crash. We consider the tree operations performed from the beginning of the ith era until (but not including) the ith crash. The linearizability arguments for these operations is analogous to the arguments established for linearizability in the OCC-ABtree: the linearization points used in this section are all analogous to the OCC-ABtree's linearization points, the p-OCC-ABtree satisfies analogous invariants, and the definition of a key being in the p-OCC-ABtree is Definition 12.2 (which is analogous to the OCC-ABtree's definition of a key being in the tree).

At the ith crash. At the time of the crash, the definition of a key being in the tree changes from Definition 12.2 to Definition 12.1. It must be shown that the keys in the tree after the crash are exactly those that were in the tree before the crash, plus any that were inserted by a simple insert linearized at the crash and minus any that were deleted by a successful delete linearized at the crash.

First consider the case when a key k *is* in the tree after a crash. That is, there exists some p-reachable leaf l such that `k = l.keys[i]` (for some index `i`).

If k was also in the tree before the crash (according to Definition 12.2), it is only correct for k to be in the tree after the crash if no delete of k linearized at the crash. This is indeed the case, since a delete of k that linearized at the crash would have set `l.keys[i]` to $\perp$ and flushed $\perp$. But, by assumption, k is in the keys array of l after the crash.

Otherwise, if k was not in the tree before the crash, it is only correct for k to be in the tree after the crash if an insert of k *did* linearize at the crash. This is true. Since k was not in the tree before the crash but l was p-reachable and contained k, the l's version must have been odd at the crash (according to Definition 12.2). Thus, there must have been an ongoing insert that inserted k at the time of the crash. Since the crash occurred after the flush of k but before the version was incremented to an even number, this insert linearized at the crash.

A similar argument shows that k is *not* in the tree after a crash if and only if k was either deleted at the crash or was not in the tree before the crash (and was not inserted at the crash).

p-OCC-ABtree invariants 1-4 and 7 are maintained during a crash since they only describe persisted data. Invariants 5 and 6 might be incorrect since they refer to volatile fields. However, they are restored by the recovery procedure.

After the ith crash (recovery). The recovery procedure does not affect the set of p-reachable nodes or their keys or values, so the set of keys in the tree is fixed while the system is recovering. By the time the recovery procedure returns, all p-reachable nodes' versions are 0, and thus the key in the tree is the same according to Definitions 12.1 and 12.2.

Additionally, all p-OCC-ABtree invariants are satisfied by the time the recovery procedure returns. p-OCC-ABtree invariants 1-4 and 7 were correct before recovery, and the recovery procedure fixes the volatile fields, which ensures that invariants 5 and 6 hold by the time it returns.

Thus, the execution up to the beginning of the operation in the $i+1$th era is strictly-linearizable, and the p-OCC-ABtree satisfies all invariants. $\square$

The proofs for the MCS-ABtree, CoMCS-ABtree, and CoPub-ABtree are similar. Note that coalescing does not conflict with the change of linearizing some operations at a crash. In both the CoMCS-ABtree and the CoPub-ABtree, an operation O_e is only coalesced *after* the successful operation O_p has executed its second increment of the leaf's version. Any simple insert or successful delete has linearized by this time (and a future crash does not change this fact).

7.4 Optimizations

There are two optimizations that can be made to the tree. Both remove the need for the first `sfence` (after writing `val`) in simple inserts. Recall from Chapter 2 that flushing multiple cache lines at an `sfence` is no more expensive than flushing a single cache line, since cache lines are flushed in parallel. That is, the number of `sfence` instructions should be considered as the performance cost of persistent memory, not the number of flushes.

Normally, in simple inserts, it is important to wait for `val` to be flushed *before* inserting `key` because `key` and `val` are not necessarily in the same cache line. Without the `sfence`, it is possible for the cache line containing `key` to be flushed asynchronously (i.e. in the background) but the cache line containing `val` to not be flushed. This results in an invalid key-value pair in the tree, which might be erroneously recovered after a crash.

The first optimization is to ensure that each key is in the same cache line as its associated value. This can be accomplished by, for example, using a key-value pairs array instead of a keys array and a values array. This was not implemented because it complicates the implementation: if internal nodes also have a keys-pointers array, searches might become slower since the keys are not as densely packed and occupy more cache lines; if not, internal nodes and leaf nodes have different memory layouts and require type information to distinguish them.

The second optimization is to use hardware that guarantees `flush` instructions are first-in-first-out (FIFO) ordered. That is, if `flush` is first performed on the cache line containing `val` then the cache line containing `key`, `val` is flushed before key.

7.5 Number of `sfence`s vs FPTree

The FPTree is an efficient concurrent, persistent B-tree with a number of optimizations (see Chapter 3). In this section, we compare the number of `sfence` instructions required in each of the operations performed by the FPTree and the p-OCC-ABtree. The FPTree does not persist its internal nodes, which might lead one to believe that it requires fewer `sfence` instructions than the p-OCC-ABtree. However, the p-OCC-ABtree only requires one `sfence` for simple inserts, whereas the FPTree uses two. Since simple inserts and successful deletes make up the bulk of the operations on the tree, the p-OCC-ABtree actually requires *fewer* `sfence` instructions. Moreover, the p-OCC-ABtree only requires two `sfence`s for splitting inserts, whereas the FPTree requires five.

7.5.1 Simple inserts

The FPTree [46] uses a bitmap to indicate whether a key-value pair is valid. The bitmap for a key is set to 1 after a key is inserted to mark it as valid, and set to 0 to mark a key as deleted. Simple inserts in the FPTree require two `sfence`s: one after inserting the key and one after changing the bitmap. In contrast, simple inserts in the p-OCC-ABtree only require one `sfence` (using one of the optimizations in the previous section).

The decision to represent an empty slot in the OCC-ABtree with $\perp$ is essentially what allows the optimized simple insert to only need one `sfence`. This might seem like a significant restriction, since it is often implemented by reserving a key that represents $\perp$. Indeed, the implementation of the OCC-ABtree from Section 6 *does* reserve a key to represent $\perp$. However, this is not the only way of representing an invalid key. $\perp$ could equivalently be represented by any key outside the key range of the leaf. `search` can easily be modified to return the key range of the leaf as well, allowing any operation to determine whether a key in the leaf represents $\perp$ or is a valid key.

7.5.2 Successful deletes

Both the p-OCC-ABtree and the FPTree only require one `sfence` for successful deletes. The p-OCC-ABtree flushes the deleted key's cache line an performs an `sfence`. The FPTree flushes the bitmap entry for the key after setting it to 0 (to indicate the key-value pair is no longer valid) and performs an `sfence`.

7.5.3 Splitting inserts

Splitting inserts in the FPTree are *not* atomic. Thus, the FPTree uses logging to ensure that it can be recovered to a consistent state after a crash. This logging approach requires five `sfence` instructions. In comparison, the p-OCC-ABtree only requires two `sfence`s: one to persist data in the new nodes and one to persist the pointer to the new nodes.

7.5.4 Rebalancing steps

Rebalancing steps are not persisted in the FPTree, but are in the p-OCC-ABtree. Like in splitting inserts, the p-OCC-ABtree requires two `sfence`s: one to persist data in the new nodes and one to persist the pointer to the new nodes.

7.6 Performance

This section briefly compares the performance of the persistent trees introduced in this thesis to their volatile counterparts.

7.6.1 Experimental setup

The experimental setup and methodology are the same as Section 6.1, except the system is a 2-socket Intel Xeon Gold 5220R CLX with 24 cores per socket and 2 hyperthreads (HTs) per core (for a total of 96 hardware threads), 192GiB of RAM, and 1536GiB of Intel 3DXPoint non-volatile RAM as the main memory.

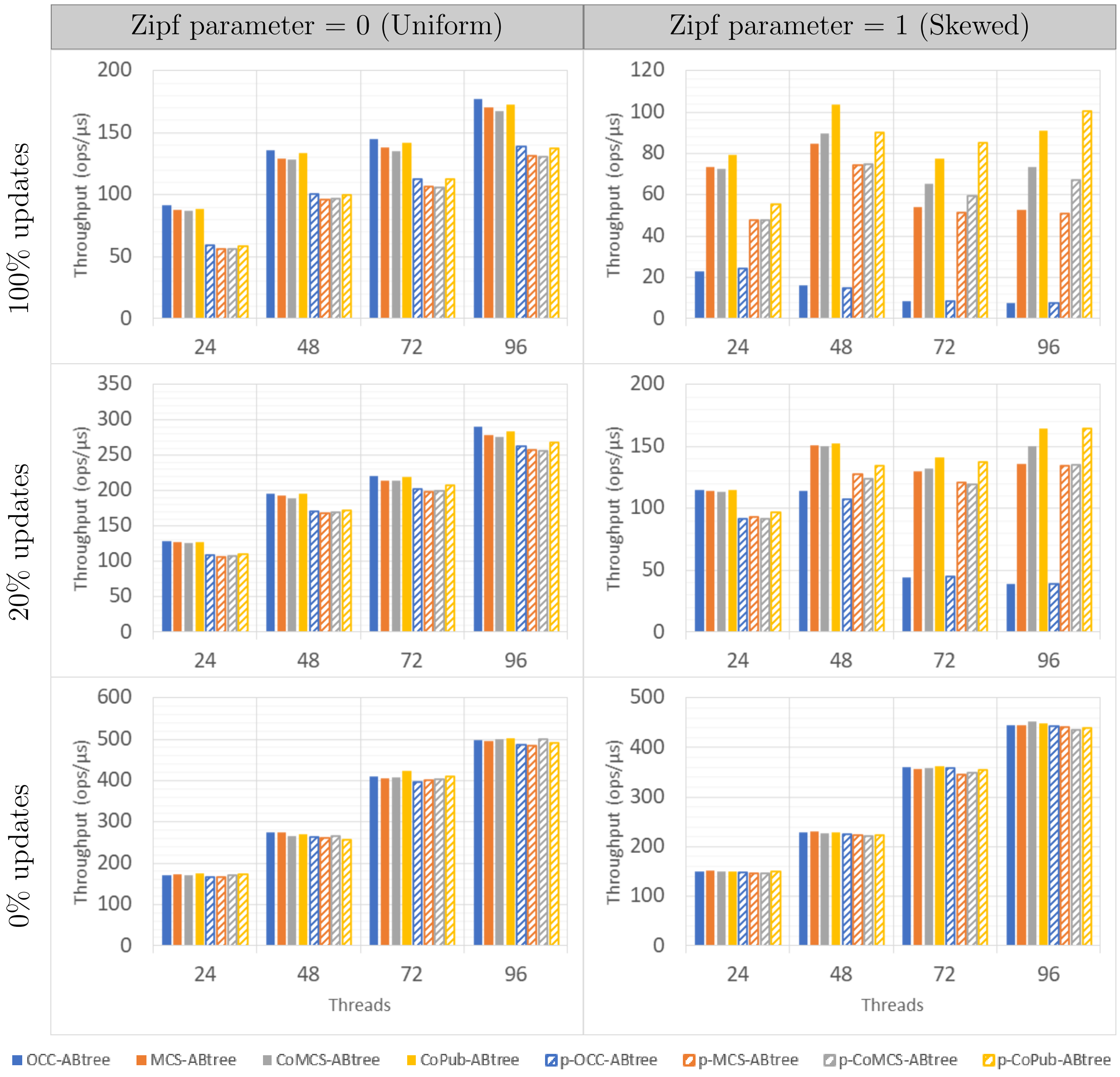

Figure 7.2: Results of benchmark with **10K keys**.

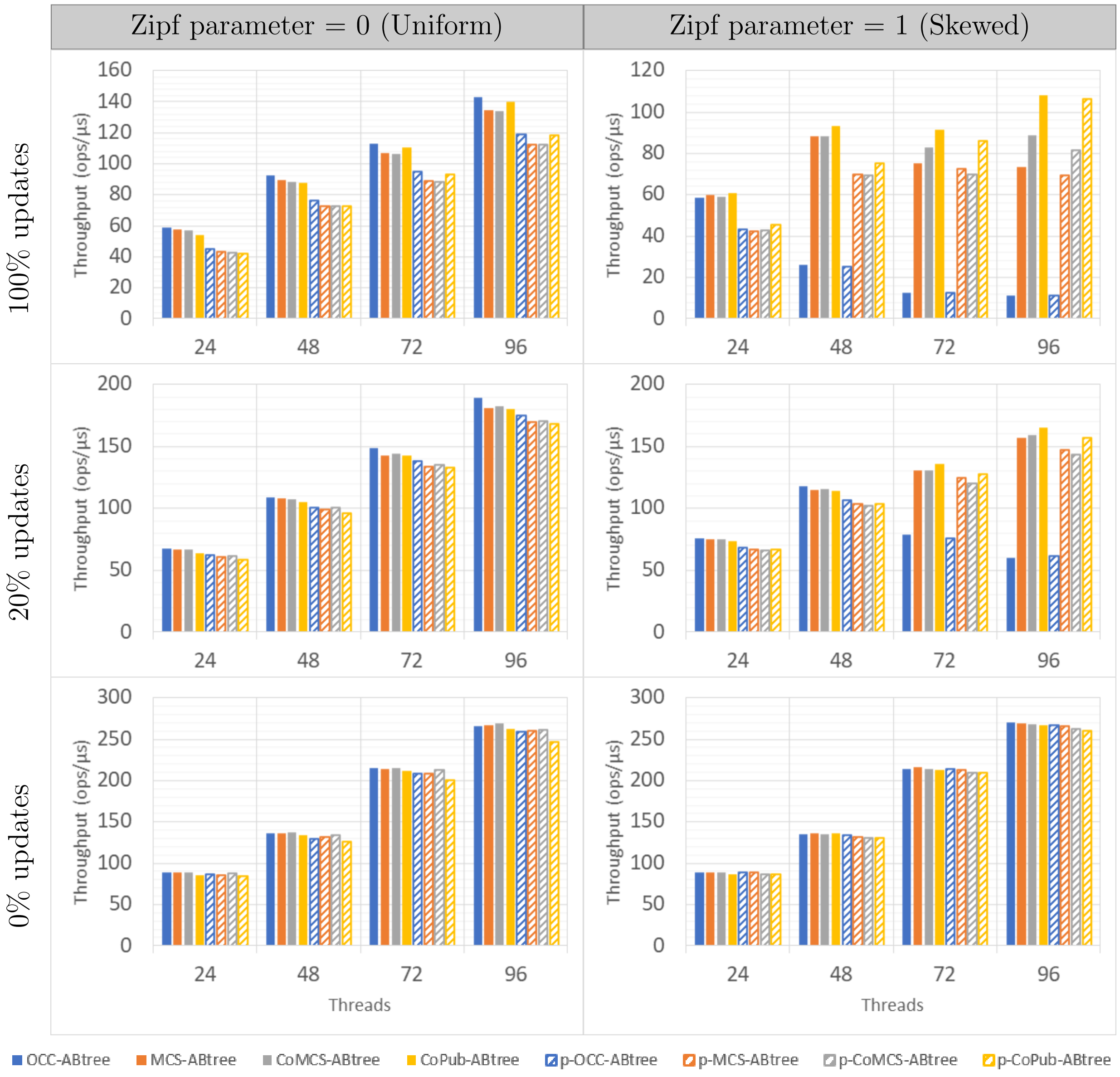

Figure 7.3: Benchmark with **1M keys**.

7.6.2 Results discussion

Figures 7.2 and 7.3 contain the results of the experiments for 10000 and 1 million keys, respectively.

Considering only the persistent trees, we see trends that match the trends from the volatile trees. On uniform and search-only workloads, the persistent trees all perform roughly the same.

In the workloads with contention (20% and 100% updates with a Zipfian access distribution), the p-OCC-ABtree scales far worse than the other persistent trees, especially on multiple processor sockets (e.g. the sharp decrease in performance in the 20% workload with Zipfian access). This is likely due to the cost of cache coherency, as it was for the volatile trees. Coalescing appears to increase the performance of the persistent CoMCS-ABtree and CoPub-ABtree in high-contention workloads as it did in the volatile setting.

Comparing the persistent and volatile trees, we observe the following trends. As expected, the persistent and volatile memory trees perform equally well on search-only workloads (since there is no flushing). The slight decrease in performance in the persistent trees is due to the (very minor) overhead of having to verify that each pointer is unmarked (even though this is always true because the tree is not changing).

In the 20% and 100% update workloads, the added cost of persisting data makes the persistent memory trees slightly slower than the volatile memory trees overall. The most interesting feature of these workloads is the performance of the persistent CoPub-ABtree in the 10000 keys benchmark. The relative difference in performance between the persistent CoPub-ABtree and the other persistent trees is *larger* than the different between the volatile CoPub-ABtree and the other volatile trees. This is likely because coalescing (if implemented efficiently) reduces the number of flushes that have to occur (since coalesced operations do not change the tree and do not flush).

Overall, these results suggest that coalescing is a promising concept in persistent memory as well — in fact, it sometimes improves performance even more than it does in volatile memory systems.

Coalescing might also be useful for preventing wear on the persistent memory hardware. Intel Optane systems contain dedicated memory controllers for persistent memory [59]. These memory controllers handle wear-leveling and bad-block management. They also contain a number of *write-combining buffers*, which coalesce writes to the same memory address, thus reducing the hardware wear caused by multiple writes.

The coalescing used in the CoMCS-ABtree and CoPub-ABtree can be viewed as a software fallback for these buffers, similar to how software transactional memory is a fallback for hardware transactional memory. Software transactional memory allows for transactions to complete when they would normally abort in hardware transactional memory (e.g. for cache capacity reasons). Similarly, the software coalescing algorithms presented in this thesis are able to perform coalescing on any number of addresses, whereas the maximum amount of hardware coalescing is restricted by the number and size of buffers.

Chapter 8

Future work and concluding remarks

The coalescing schemes described in this thesis do not work as-is if the desired semantics of `insert` are that an existing value associated with the key should be *replaced* and the old value returned, since this requires communication between all threads that are coalescing themselves. The MCS coalescing scheme could be modified to support these semantics by maintaining the current value associated with the key as the queue is being traversed, though this would require coalesced operations to wait for coalescing to complete before returning, which might be slower. In general, the MCS queue could be used to implement flat combining [29] to combine more general operations (e.g., incrementing a counter) into a single write.

The simple change of using MCS locks drastically improved the performance of the OCC-ABtree. Using NUMA-aware locks like HCLH [38], lock cohorting [22], or NUMA-aware reader-writer locks [15] might also be a simple way of improving performance further.

Several of the optimizations from existing persistent trees could be applied to the persistent trees in this thesis. The fingerprints used in the FPTree could be used to quickly compare keys with expensive comparisons (e.g. strings). The wB$^+$-Tree is a sequential persistent B-tree in which leaves contain a *slot array* storing the sorted order of key slots [17]. This allows for binary search within a leaf, at the cost of having to update the slot array on inserts and deletes. This optimization might not be effective on high-update workloads of this thesis. The clfB-tree is another sequential persistent B-tree which uses *differential encoding* of keys and pointers to fit more keys into a node [17]. The first key in a node is stored normally, but every subsequent key is stored as the difference between it and the *previous* key. Since keys in a node are usually close to each other, this optimization can greatly reduce the number of bytes required to store a key (and thus increase the number of keys that can be stored per node). A similar optimization can be applied to pointers. The main drawback of differential encoding is that searches must decode the keys/pointers, which can take a significant amount of time.

Finally, testing the persistent (a,b)-trees using an allocator designed for persistent memory and comparing against existing persistent concurrent B-trees would be interesting. Unfortunately, there do not appear to be any official public implementations of the three persistent concurrent B-trees mentioned in Chapter 3.

In total, this thesis introduced eight different (a,b)-trees, each excelling in some setting.

The OCC-ABtree has the best performance out of all the trees under uniform workloads. The MCS-ABtree scales well, even under Zipfian, moderate-update workloads. The CoMCS-ABtree and CoPub-ABtree are able to use contention to their advantage and drastically outperform existing data structures on high-contention workloads, while retaining almost all of their low-contention performance. And, the persistent versions of these trees each provide their respective advantages in the increasingly common persistent memory setting.

Hopefully, the success achieved in each of these domains convinces the reader that investing in workload-specific improvements for concurrent algorithms can yield significant performance gains.

References

[1] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming.* Advanced Micro Devices Inc, March 2021.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

[3] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. Cbtree: A practical concurrent self-adjusting search tree. In *Proceedings of the 26th International Conference on Distributed Computing*, DISC'12, page 1–15, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

[5] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. The Splay-List: A Distribution-Adaptive Concurrent Skip-List. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[6] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 14–27, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 295–306, Boston, MA, July 2018. USENIX Association.

[8] 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.

[9] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 257–268, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Trevor Brown. *Techniques for Constructing Efficient Lock-free Data Structures.* PhD thesis, University of Toronto, 11 2017.

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, page 13–22, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 329–342, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 276–291, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, page 261–270, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 157–166, New York, NY, USA, 2013. Association for Computing Machinery.

[16] 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.

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

[18] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the Twenty-Second*

Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '10, page 335–344, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 336–346, New York, NY, USA, 2006. Association for Computing Machinery.

[20] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 373–385, USA, 2018. USENIX Association.

[21] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 631–644, New York, NY, USA, 2015. Association for Computing Machinery.

[22] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 247–256, New York, NY, USA, 2012. Association for Computing Machinery.

[23] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, page 131–140, New York, NY, USA, 2010. Association for Computing Machinery.

[24] Panagiota Fatourou and Nikolaos D. Kallimanis. The redblue adaptive universal constructions. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC'09, page 127–141, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 325–334, New York, NY, USA, 2011. Association for Computing Machinery.

[26] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[27] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition.* Morgan and Claypool Publishers, 2nd edition, 2010.

[28] William Hasenplaugh, Andrew Nguyen, and Nir Shavir. Quantifying the capacity limitations of hardware transactional memory. In *PODC '15: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[30] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, page 313–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] 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.

[32] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, page 161–171, New York, NY, USA, 2012. Association for Computing Machinery.

[33] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, volume 9888, pages 313–327, 09 2016.

[34] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. Clfb-tree: Cacheline friendly persistent b-tree for nvram. *ACM Trans. Storage*, 14(1), February 2018.

[35] Kim S. Larsen and Rolf Fagerberg. B-trees with relaxed balance. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, page 196–202, USA, 1995. IEEE Computer Society.

[36] 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.

[37] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313, April 2013.

[38] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical clh queue lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, page 801–810, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] Karl Malbrain. A blink tree latch method and protocol to support synchronous node deletion, 2014.

[40] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

[41] Marlon McKenzie. Creating a concurrent in-memory b-tree optimized for numa systems. Master's thesis, University of Waterloo, 9 2015.

[42] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[43] Wojciech Mula. Skip list.svg, 2008. https://commons.wikimedia.org/wiki/File:Skip_list.svg.

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

[45] Aravind Natarajan, Lee H. Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 8255*, SSS 2013, page 45–60, Berlin, Heidelberg, 2013. Springer-Verlag.

[46] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, page 348–354, New York, NY, USA, 1984. Association for Computing Machinery.

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

[49] Arunmoezhi Ramachandran and Neeraj Mittal. A fast lock-free internal binary search tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, 2019.

[51] K. Sagonas and K. Winblad. Contention adapting search trees. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 215–224, 2015.

[52] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.

[53] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[54] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[55] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[56] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, April 2018.

[57] Ziqi Wang. Consistent and durable data structures for non-volatile byte-addressable memory. [paper review], 2019. https://wangziqi2013.github.io/paper/2019/08/28/cdds.html.

[58] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 473–488, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182, 2020.

[60] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems.

In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 167–181, USA, 2015. USENIX Association.