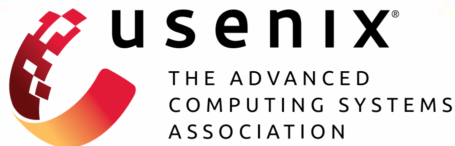


Occualizer: Optimistic Concurrent Search Trees From Sequential Code

Tomer Shanny and Adam Morrison, *Tel Aviv University*

https://www.usenix.org/conference/osdi22/presentation/shanny

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Occualizer: Optimistic Concurrent Search Trees From Sequential Code

Tomer Shanny
Tel Aviv University

Adam Morrison
Tel Aviv University

Abstract

This paper presents Occualizer, a mechanical source code transformation for adding scalable optimistic synchronization to a sequential search tree implementation. Occualizer injects synchronization only to the update steps of tree operations, leaving traversal steps to execute unsynchronized, thereby maximizing parallelism.

We use Occualizer to create concurrent versions of a sequential B+tree, trie, and red-black tree. Evaluation on a 28-core machine shows that Occualizer's trees significantly outperform prior mechanically-crafted trees on non-read-only workloads and are comparable (within 4%) on read-only workloads. Overall, Occualizer shrinks the performance gap between mechanically- and hand-crafted trees by up to 13×. When using Occualizer's B+tree as the index in the STO main-memory database, the system's throughput degrades by less than 30% compared to the default Masstree index, and it scales better.

1 Introduction

In-memory tree data structures, or *search trees*, lie at the foundation of many systems, from databases [30, 58–60, 80] through operating systems [19–21] to storage engines [46, 68, 71]. Performance in such multicore systems depends not only on the sequential (single-threaded) speed of searching the tree, but also—often, mostly—on the scalability of the tree's *synchronization protocol*, which ensures correctness of concurrent tree operations [29, 47].

Scalable synchronization protocols typically apply *optimistic concurrency control* (OCC). In an optimistic protocol, traversals of tree paths are read-only and do not perform synchronization such as acquiring locks or executing atomic read-modify-write (RMW) instructions [11, 16, 29, 71]. Synchronization occurs only if and when an operation starts updating the tree. The optimistic approach thus limits serialization of tree operations (due to locking and/or cache coherence contention) mostly to the step that physically mutates the tree, allowing other steps to execute completely in parallel. The result is scalable performance that improves as the amount of hardware parallelism grows (unless the workload is contended at the semantic level, e.g., operations updating the same key).

Deploying an optimistic concurrent search tree in a system can be a hard problem, however. Systems often cannot deploy "off the shelf" trees, as their target use cases and workloads call for new, customized data structures [6, 17, 65, 67, 71, 80]. But designing a scalable synchronization protocol for a custom data structure—particularly an optimistic protocol—is notoriously challenging, because it involves *concurrent reasoning* to verify the algorithm's correctness under any possible thread interleaving allowed by the protocol [55, 64]. This effort also needs to be repeated whenever the data structure's algorithm changes, e.g., due to new optimizations or features.

To solve the problem of manually adding synchronization to a data structure, concurrency research has proposed *automatic transformations* such as universal constructions [2, 3, 18, 26, 35, 40, 51, 52] and transactional memory [54, 77]. These transformations receive a sequential data structure implementation (code) and produce a correctly synchronized version.

When applied to search trees, however, the automatic transformations do not produce efficient, scalable data structures. Some transformations inject pessimistic synchronization, which fully serializes all operations [18, 40, 51, 52] or all non-read-only operations [5, 26, 35]. Transactional memory-style transformations [2, 3, 31, 35, 41, 77] use optimistic synchronization, but block or restart operations whose path crosses nodes modified by a concurrent update operation, which degrades scalability. Overall, current automatic transformations produce trees whose throughput flatlines beyond 12 cores if even 3% of the workload's operations are not lookups (§ 7), as typically happens in dynamic workloads [4, 20, 58, 71].

Solution: Occualizer. This paper proposes *Occualizer*, a mechanical transformation for augmenting common sequential search tree implementations with scalable optimistic synchronization,[1] producing linearizable [56] concurrent trees. Occualizer's transformation requires the input tree to satisfy certain natural prerequisites, which most algorithms we are aware of meet, and our current prototype requires some manual effort to transform the input code. Occualizer injects synchronization only to the update steps of an operation (if any), leaving traversal steps to execute unsynchronized, unchanged from their baseline sequential code. Our key idea is to design Occualizer's injected synchronization so that it satisfies the "forepassed" condition of Feldman et al. [44]—which they prove implies the correctness of unsynchronized traversals in the presence of concurrent updates. We thus *design synchronization to satisfy a proof* instead of endeavoring to find a proof for our synchronization.

[1] Occualizer: one that adds OCC (optimistic concurrency control).

Informally, the "forepassed" condition requires that if a memory write w in the concurrent tree changes the search path for a key k, then any node v removed from the path must become immutable [44].[2] To obtain this property, Occualizer uses *localized copy-on-write* (LCOW), wherein all of an operation's writes are performed by atomically replacing the written-to nodes with new, updated copies, and making the old copies immutable. Crucially, LCOW does not require copying the entire path from the root to the updated nodes and thereby avoids synchronization bottlenecks at the top of the tree—a fundamental difference from prior COW techniques [5, 19].

We use Occualizer to produce concurrent versions of the sequential `tlx` B+tree [8], a radix tree (trie), and a red-black tree, and evaluate them on a dual-socket 28-core machine. Compared to prior transformations, Occualizer's search trees are far faster and more scalable in dynamic (non-read-only) workloads, outperforming trees using GCC's transactional memory by up to 17× and the CX universal construction [26] by orders of magnitude. On read-only workloads, Occualizer's trees are comparable to prior constructions' (within 4%). Due to instrumentation overheads, however, Occualizer's trees do not match the performance of hand-crafted concurrent algorithms. For instance, when used as an index in the STOv2 main-memory database system [58], Occualizer's B+tree has better scalability but is 25%–30% slower than the default index, Masstree [71], a hand-crafted concurrent trie/B+tree hybrid.

Overall, Occualizer significantly changes the cost/benefit analysis of hand-crafting a concurrent search tree. By shrinking the performance gap between mechanically- and hand-crafted trees by up to 13×, Occualizer makes mechanically-crafted trees applicable in many more contexts and performance targets, freeing up time and costs that would otherwise be spent on designing, implementing, and testing a hand-crafted implementation.

Contributions. We make the following contributions:

- **Transformation.** We describe Occualizer, a mechanical transformation for augmenting common sequential search tree implementations with optimistic synchronization.

- **Implementation.** We implement Occualizer and use it to produce concurrent versions of the sequential `tlx` B+tree, a radix tree (trie), and a red-black tree. Occualizer's code is available at `https://github.com/tomershanny/Occualizer`.

- **Evaluation.** We show that Occualizer's trees outperform trees using GCC's transactional memory by up to 17× and the CX universal construction by orders of magnitude, but are slower than hand-crafted concurrent trees.

[2] Intuitively, this condition guarantees that any operation whose search for k is currently located at v will either rejoin the new path or will end at an immutable node from which it cannot "damage" the tree.

2 Background, motivation, and related work

Designing efficient fine-grained synchronization for data structures is notoriously hard, because verifying synchronization correctness requires reasoning about every possible thread interleaving allowed [55, 64], while scalability requires the protocol to allow more possible interleavings. Optimistic search tree design exemplifies this challenge. On one hand, to maximize scalability, the synchronization protocol should not block or restart a traversal that encounters concurrent updates of its search path [11]. On the other hand, a traversal encountering such updates can observe *inconsistent* tree states, which cannot occur in a sequential execution but must be reasoned about to verify the protocol's correctness [43, 44, 61, 62, 66, 73, 81, 82].

The difficulty of designing a highly-scalable and correct optimistic tree lead some systems to deploy search trees with relaxed correctness guarantees. Linux's red-black tree, for instance, guarantees only that searches do not crash in the face of concurrent updates—but not search correctness [69]. Researchers have identified principles for designing optimistic synchronization protocols [11] as well as compiler support to simplify their implementation [84], but such research does not address the fundamental verification difficulty of a scalable, human-designed synchronization protocol.

Our motivation is therefore to automate the task of adding optimistic synchronization to a custom-designed sequential search tree. Concurrency research has proposed approaches for automatically transforming a sequential data structure into a concurrent one: universal constructions (§ 2.1) and transactional memory (§ 2.2). But these approaches do not produce scalable concurrent data structures when applied to search trees, as we discuss next.

2.1 Universal constructions

A *universal construction* (UC) [51] takes a sequential implementation of a data structure and outputs a linearizable concurrent version of it, without modifying the sequential code—i.e., by "wrapping" it in synchronization in some fashion. UCs can apply *nonblocking* or *blocking* synchronization.

Nonblocking universal constructions create concurrent data structures with nonblocking progress properties: either *wait-free*, which means every operation can complete in a finite number of its own steps, or *lock-free*, which means that some operation always completes after a finite number of execution steps [51]. Achieving these progress guarantees typically requires operations to coordinate and *help* each other make progress, which adds overhead [55].

Blocking universal constructions are based on the *delegation* technique [10, 15, 39, 50, 70, 74], which delegates the execution of the data structure operations threads to one thread. This "server" thread executes operations on behalf of the other, "client" threads. Delegation schemes differ in the types of operations delegated (e.g., all operations [39, 50, 70], up-

date operations [15], or read-only operations [10]) and/or their inter-thread communication techniques [14, 75].

The flip side of UCs' treatment of the input code as a black box is that the synchronization they add is coarse-grained, pessimistic, and slow. Early nonblocking UCs [18, 38, 40, 51, 52] work by having each operation execute on a local copy of the entire data structure which it then tries to install as the new version. This approach fully serializes all operations and is prohibitively slow on real-world-sized data structures. Early delegation UCs [39, 50, 70] also fully serialize all operations, as they delegate every operation to the "server" thread. Modern nonblocking UCs [5, 26, 35] improve on the full serialization aspect by optimizing read-only operations, allowing them to execute in parallel, but non-read-only operations are still serialized. Likewise, modern delegation UCs allow certain types of operations to execute in parallel (updates [10] or read-only operations [15]), but all other operations remain serialized by delegation.

COW UC. A nonblocking UC technique that reduces copying overhead (and is closely related to Occualizer) is copy-on-write (COW), used in the transactional system of Ben-David et al. [5]. The core idea is for a writing operation to create its updated version without copying the entire data structure, by having it share as much as possible with the previous version. For trees, this technique updates a node by creating an updated version of the node and the path that leads to it, and atomically swapping the updated root with the new one (Figure 1). The COW approach allows read-only operations to proceed without synchronization, since the version they observe is immutable. Writing operations, however, remain serialized.

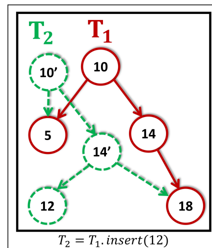


Figure 1: COW: Adding 12 as a child of 14 in T_1 yields T_2.

2.2 Transactional memory

Transactional memory (TM) [54, 77] executes sequential code segments as isolated atomic transactions. With hardware TM (HTM), serializability of the transactions is enforced by the hardware [54]. HTM can thus be viewed as a UC. Real-world HTM extensions, however, have several limitations [33, 33, 34] and are currently disabled on many processors due to hardware errata [63]. We therefore focus on software TM (STM). STMs differ from UCs in that they require *code instrumentation*, so that the STM runtime can mediate reads/write to memory and (in some cases) memory allocation/deallocation.

Modern STMs have converged on designs using optimistic-style lock-based synchronization [36, 48]. In these designs, the STM algorithm performs transactional reads without writing to memory (e.g., to acquire a lock); writes either acquire locks ("eager" locking) or are buffered in a write set ("lazy" locking). When the transaction ends, the STM checks whether there is a point in time in which all of the transaction's reads and writes can appear to take place atomically. If so, the transaction *commits* and its writes are made visible to other transactions (e.g., locks are released). Otherwise, the transaction *aborts* and must restart.

Unfortunately, since the STM does not understand the semantics of the underlying code, its validation conservatively depends on every value read by the transaction [31, 41]. Therefore, if any memory location read by a transaction is written to before the transaction commits, the transaction will abort. This effect severely limits scalability of STM-based trees, because any concurrent write to an operation's search path causes the operation to abort—even if the operation would have reached the same location in the tree had it executed on the new path (a fact the STM cannot know). In our experiments, TM performance can flatline at low core counts even if as few as 3% of the tree operations are updates (§ 7).

TM research has proposed several approaches to address the above problem. First, an STM can determine the serial order of transactions (*conflict detection*) more intelligently [76, 85]. But this typically requires transactional reads to write to memory, which can lead to undesirable serialization of readers. Second, transactions can be built over higher level objects instead of low-level memory reads/writes [49, 53, 57]. But this requires designing the underlying thread-safe objects, which was our original problem. Finally, transactional semantics can be relaxed [42] to avoid aborting a transaction in cases such as search path changes. But then one has to prove the resulting relaxed transactions correct, which requires the concurrent reasoning about thread interleaving that we wish to avoid.

2.3 Summary and goals

In summary, there is still no mechanic way to transform the source code of a sequential tree implementation into an optimistic, fast, and scalable concurrent tree—without needing to perform concurrent reasoning to verify the correctness of the produced concurrent code. This is our goal.

Occualizer sits in the middle between UCs and TM. Compared to universal constructions, Occualizer takes a pragmatic approach. Instead of accepting arbitrary sequential code as input, Occualizer requires the input to have certain natural prerequisites, and also transforms/instruments the sequential code. Compared to transactional memory, Occualizer is specialized to search trees, which enables us to design an optimistic synchronization protocol that does not restart operations whose search path is modified by concurrent operations.

3 Occualizer Overview

Occualizer receives source code of a sequential (single-threaded) search tree and transforms it into an optimistic concurrent implementation by adding calls to Occualizer's synchronization library into the input source code. This section gives an overview of the Occualizer transformation.

We first define the family of sequential tree implementations to which Occualizer is applicable (§ 3.1). (We discuss verifying that a sequential tree meets Occualizer's requirements in § 3.5.) We then give an overview of Occualizer's source code changes (§ 3.2) and the run-time synchronization protocol they inject, called localized copy-on-write (§ 3.3). The details are described in §§ 4–5. Finally, we outline the correctness proof of the produced concurrent tree (§ 3.4), which appears in § 6.

3.1 Scope

We consider correct sequential implementations of a dictionary datatype, namely, that provide lookup, insert, and delete operations on key-value pairs. The implementation may also (optionally) support ordered iteration over the stored keys, provided via key predecessor/successor operations. We assume a programming language with manual memory management. (Our prototype targets C++.)

Occualizer requires the sequential input algorithm to meet certain prerequisites (PRs), detailed below. At a high level, the prerequisites are that (1) tree operations are composed of a read-only traversal followed by reads and writes which are determined only by what the operation observes after the traversal; (2) each step in the traversal depends only on the target key and the current node; and (3) any operation that moves a node v off some search path(s) must also access v.

The user is responsible for verifying that the input meets Occualizer's prerequisites, and the concurrent tree produced by Occualizer is not guaranteed to be correct if they are not met. The human effort required for this verification (and the possibility of errors there) are limitations of Occualizer compared to general universal constructions that accept arbitrary code. While our experience has been that the prerequisites are met by many algorithms and that verifying them requires reasonable effort (see § 3.5), our vision is to develop automated verification of the prerequisites to fully automate Occualizer.

Prerequisites. We define the prerequisites in terms of an algorithm maintaining a directed graph G of nodes, whose edges represent pointers between nodes.

PR1 Maintain a rooted tree: At the end of any sequence of operations, the graph G is a rooted tree.

Crucially, PR1 does not care about intermediate states that occur while a tree operation executes, only about the graph's structure upon its completion. PR1 is conservative, as Occualizer can support structures with auxiliary edges linking nodes to their successor/predecessor, which create multiple paths from the root to nodes and so are not formally trees. We defer these details to § 4.

PR2 Read-only traversals: Every operation $op(k)$ consists of a read-only traversal *traverse*(k) that searches for k followed by read/write steps.

PR2 is not met by self-balancing trees that perform balancing during traversals, such as splay trees [78]. But PR2 is met by self-balancing trees such as red-black, AVL, or B-trees, which perform self-balancing after updating the tree (post-traversal).

PR3 Traversals are single-step: The next node visited by *traverse*(k) depends only on k and on the current node.

PR3 is met by trees with comparison-based traversals, such as B+trees [22], Bw-Trees [67, 83], red-black and AVL trees, etc., where how *traverse*(k) proceeds depends only on how k compares to the key(s) of the current visited node. PR3 can also be met by tries, provided that nodes encode the key offset they represent; otherwise, the next node visited becomes dependent on all the nodes visited so far, which violates PR3.

Our next prerequisite states that the reads and writes an operation performs after its traversal are not a function of observations made during the traversal:

PR4 Post-traversal actions depend only on subgraph accessed post-traversal: Consider an operation $op(k)$ that executes on tree T, whose *traverse*(k) finishes at node v. Let $RW^{op(k)}$ be the set of nodes read/written to by $op(k)$ after finishing its traversal. Let $T_{RW^{op(k)}} \subseteq T$ be the smallest subgraph of T containing $RW^{op(k)}$. Then for any sequence of operations that execute on T resulting in tree T', if $T_{RW^{op(k)}} \subseteq T'$ and *traverse*(k) executed on T' finishes at the same node v as in T, it holds that running $op(k)$ over T' results in exactly the same reads and writes as in op's execution over T.

PR4 does not preclude an algorithm from reading or writing parts of its search path after completing the traversal—as in, e.g., rebalancing of red-black, AVL, and B-trees—because any node on the path that an operation $op(k)$ reads or writes post-traversal becomes part of $RW^{op(k)}$. PR4 is thus met by classic tree algorithms that perform post-traversal rebalancing.

PR5 Moving off a search path implies a post-traversal access: For any operation op and any key k, consider the paths P and P' that would be taken by *traverse*(k) before and after op executes. Then if $v \in P$ but $v \notin P'$, op must read, write, or destroy v after its traversal.

For an implementation that does not expose nodes to its client, PR5 is met by every tree algorithm we are aware of. In these algorithms, a node moves off a search path due to either (1) a structural modification that changes the node's position in the tree, in which case the node is written and/or read; or (2) being removed from the tree, in which case the implementation destroys and frees the node, as it has no other references.

Occualizer for managed languages. Occualizer's design and its prerequisites are programming language agnostic. Most tree implementations, however, fail to meet PR5 when implemented in a managed language. The reason is that

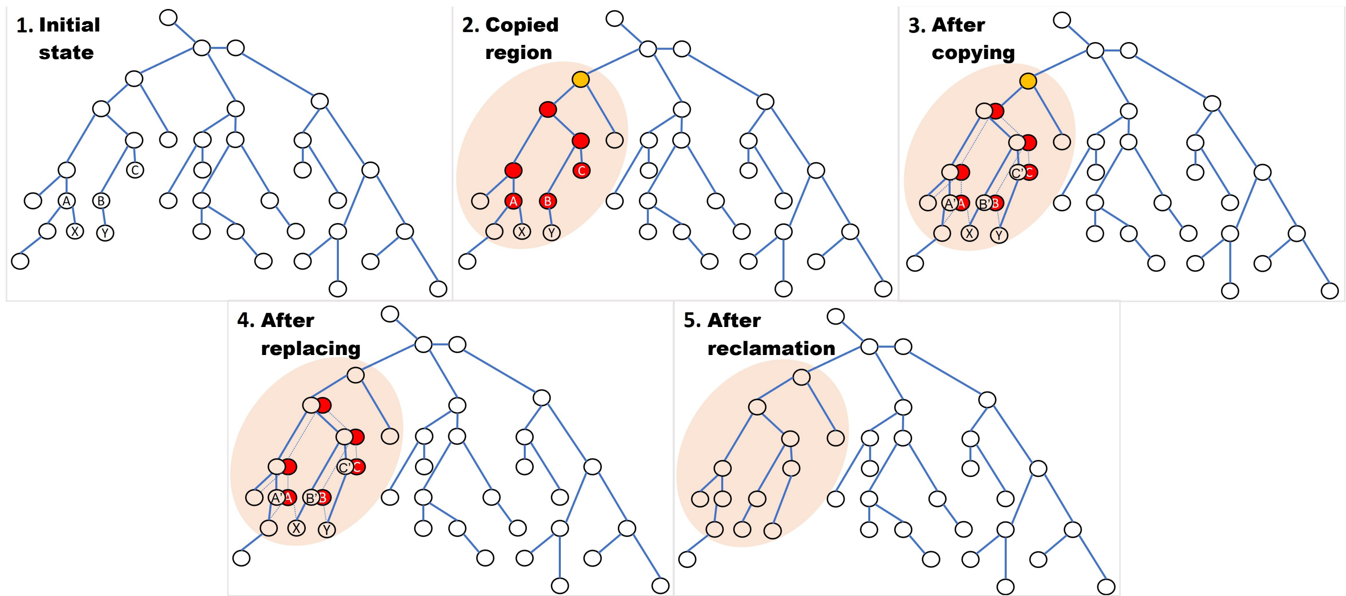


Figure 2: Illustration of LCOW on an operation that makes B and C the parents of X and Y, respectively.

node destruction in managed languages is performed asynchronously by the garbage collector and not explicitly by the program, so a managed tree implementation meets PR5 only if it writes to a node when removing it from the tree—which most algorithms do not do. This problem can be fixed (currently, manually) by adding no-op writes to removed nodes.

3.2 Code transformations

Occualizer transforms the sequential input code by adding calls to Occualizer's synchronization library. These calls are similar to object-level transactional memory instrumentation. They include calls to demarcate each operation's start and completion, and to access (read/write) node fields only through the library's interface. Field accesses are captured straightforwardly by requiring the sequential input code to access fields using only getter/setter methods, which the transformation then replaces. The transformation also adds locks and metadata fields to the node structure.

The code transformation is mechanic and our design is for it to be done automatically, with minimal user involvement. In our current prototype, however, we implement only the synchronization library and perform the code transformations of the evaluated trees manually (following the mechanical recipe given in § 4). Implementing the automatic code transformation is an ongoing effort.

3.3 LCOW synchronization library

Occualizer's code transformation leaves the logic of traversals unchanged. In particular, traversals do not block or retry mid-operation. The synchronization added to writing operations guarantees the correctness of both traversals and writing operations. To this end, the library uses a technique we call *localized copy-on-write* (LCOW). LCOW exposes all of an operation's writes atomically, using one atomic write. Unlike other COW techniques [5, 19], this write *does not* typically target the tree's root and thereby avoids creating a synchronization bottleneck.

LCOW works as follows. Once an operation *op* finishes its traversal, the library uses a combination of locking and validation checks to maintain an invariant that *op*'s further observations of the tree are consistent with some sequential execution. This invariant is needed to guarantee that *op*'s code behaves correctly. In particular, whenever *op* first writes to some node v, the library locks v and creates a copy of v, v'. (If a lock acquisition fails, *op* is restarted, releasing any locks it holds and freeing node copies it had made.) Subsequently, all of *op*'s accesses to v are redirected to v', ensuring *op* "sees its own writes." When *op* completes, the library identifies a minimal subgraph containing all written nodes, called the *copied region*. This subgraph is itself a tree rooted at some node n, but it may not be n's subtree (i.e., it may not include all of n's descendants). Next, the library locks and creates a copy of the copied region, updated to contain the nodes written to by *op*. Finally, the library exposes *op*'s writes atomically by linking u', the root of the copied region, instead of its original version u with one atomic write. Crucially, the old versions of the nodes remain locked, making them immutable.

The library reclaims the memory of the old copied region only once it is guaranteed that no concurrent operation may be accessing the old region, using a read-copy update (RCU) epoch-based memory reclamation scheme [45, 72].

Figure 2 illustrates LCOW on some abstract operation *op*. ① shows the initial tree state. Assume that executing *op*'s sequential code from start to finish in this state would make

B and C the parents of X and Y, respectively. ② shows the copied region: LCOW locks the orange and red nodes, thereby blocking concurrent modifications to every node in the pink circle. ③ LCOW copies the red nodes, creating a new version of the copied region in which *op*'s writes are made to the nodes A', B', and C'. Finally, ④ shows the memory state after atomically replacing the copied region with its new version, and ⑤ shows the memory state after the original copied region is reclaimed.

The upshot is that Occualizer guarantees that (1) executing operations run correctly, as they would in a sequential execution, and (2) the state of the tree in memory is always a state that can be produced by a sequential execution. Crucially, however, this is all achieved while still allowing the *traversal* part of an operation to observe an inconsistent state. E.g., a traversal can start in tree T_1 and then cross into tree T_2, walking a path that never actually existed in memory.

3.4 Linearizability argument

Trees produced by Occualizer are linearizable [56], i.e., operations appear to execute atomically. The main challenge of proving linearizability is that because traversals are unsynchronized and can observe inconsistent tree states, it is not clear that a traversal ultimately reaches the correct node.

The key idea of Occualizer is to design its synchronization to satisfy the precondition of an existing proof (from the concurrency literature) that an unsynchronized traversal is correct. Occualizer's trees satisfy the "forepassed" condition, which Feldman et al. [44] prove implies that if an unsynchronized traversal searching for key k reaches node v, then at some point during its execution, v was on the search path for k. (That is, the state of the tree was such that had the traversal executed from start to finish then, it would have reached v.)

The above immediately proves the linearizability of read-only lookups that consist only of traversals. To prove linearizability of writing operations, we show that when a writing operation atomically performs its writes using LCOW, then the state of the tree is such that had the operation's sequential code executed atomically now, it would have behaved exactly the same. In other words, the state of the tree in memory remains consistent with some sequential execution of the original sequential code. We show this by first proving an invariant that an operation locking node v implies that v is on the relevant search path at lock acquisition time ("now"). The proof then follows from PR4, since the copied region locked by an operation contains the subgraph it accessed post-traversal.

3.5 Discussion: Prerequisite verification

For our evaluation (§ 7), we use Occualizer on sequential implementations of classic tree algorithms, such as the B+tree. We draw on this experience to discuss the effort and reasoning needed to manually verify that an implementation meets Occualizer's prerequisites. In a nutshell, we find that the prerequisites are met by many algorithms (e.g., red-black and B-trees) and tree design techniques. We also find that checking the prerequisites requires reasonable effort, given basic understanding of the input tree's algorithmic properties. In particular, there is no need for concurrent reasoning, as the prerequisites are properties of sequential code.

Verifying PR1–PR3 involves straightforward code inspection. In particular, verifying that every tree operation begins with a read-only traversal (PR2) is easy for implementations with an explicit traversal method and for recursive implementations, where one only needs to check the recursive function.

Verifying PR4–PR5 requires reasoning about the principles driving the sequential input algorithm. To verify that post-traversal actions depend only on the subgraph accessed post-traversal (PR4), we need to check that the nodes and fields an operation chooses to access and the values it writes depend only on what it reads after its traversal. PR4 would be violated, for example, by an operation writing a node's depth (distance from the root) that was computed while searching for the node. On the other hand, PR4 is satisfied by an operation maintaining the height of a node (or the balance factor in an AVL tree) using a bottom-up computation after the traversal. PR4 holds trivially if traversals are performed as a subroutine call that returns only the target node, thereby making the search path opaque to the operation.

To verify that if a node stops being on the search path for some key k, then the node must be accessed or destroyed (PR5), we need to verify that a node removal destroys it, and to reason about how tree structure modifications affect the behavior of searches. We find that common tree algorithmic techniques meet PR5. For instance, consider a binary tree rotation [25] moving node y above its parent y (Figure 3). The only node that moves off a search path as a result of the rotation is y (which moves off the paths leading to subtree A) and y is indeed written by the rotation (its left child changes).[3] As another example, in a binary tree that deletes an internal node by replacing it with its successor [25] (the leftmost node of its right subtree), the nodes on the path to the successor move off the search path to the successor. These nodes are read by the removing operation as it searches for the successor, so PR5 is met.

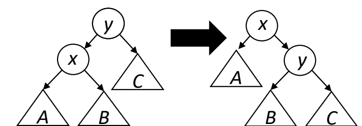


Figure 3: Rotation moving x above y.

4 Design

This section describes Occualizer's design. We first describe Occualizer's synchronization library interface and the mechanical rules for calling its methods from a sequential tree implementation (§ 4.1). We next describe how the library

[3]Crucially, PR5 depends only on the effect that a complete rotation has on future searches—not on the exact order of writes performing the rotation in the sequential code.

implements the LCOW synchronization protocol (§ 4.2) and then extend the design to support algorithms with auxiliary edges between nodes, which are typically used to optimize iteration over nodes (§ 4.3).

4.1 Library interface & code transformations

Interface. Occualizer augments the input with calls to its library. Table 1 describe the library's transactional memory-like interface, which consists of "macro" and "micro" methods.

The "macro" methods demarcate the points in which the operation starts/finishes and where its traversal ends. In particular, `occ_start` checkpoints the calling thread's register state (e.g., with `setjmp()`) and restores it if the library decides to abort the operation. When an abort happens, `occ_start` returns a failure indication.

The "micro" methods read and write node fields (or the root pointer) and notify the library of allocated or destroyed nodes. For simplicity, we show the read/write methods as taking the field name as an argument. An implementation either generates specific methods for each field or has a general method that takes the field's offset and size in the node.

Transformation. Transforming a sequential tree to an optimistically-synchronized one using Occualizer requires two types of transformations. Macro transformations add macro calls to demarcate each operation with `occ_start`, `occ_traverse_done`, and `occ_finish` calls, and restart it after an abort (Listing 1). Occualizer does not require modifying the input code to separate the traversal into its own method, only to call `occ_traverse_done` when it is done. This property allows the operation's subsequent code to reuse information learned during the traversal, e.g., to climb back up the path for tree maintenance.

Micro transformations replace calls to node setter/getter methods with the appropriate `occ_set`/`occ_get` calls, and

Method	Called when (and why)
`occ_start`	Operation starts (to initialize bookkeeping data)
`occ_traverse_done`	Traversal finishes (to start consistency checks)
`occ_finish`	Operation finishes (to atomically perform operation's writes)
`occ_restart`	Restarting an aborted operation (to free resources acquired during the failed execution)
`occ_set`(n, f, v)	Writing $n.f \leftarrow v$ (to lock and copy n)
`occ_get`(n, f)	Reading $n.f$ (to read from n's copy, if it exists)
`occ_node_born`(n)	Node is allocated
`occ_node_dies`(n)	Node is destroyed

Table 1: Occualizer synchronization library interface.

```
Function transformed<op>(args):
    while True do
        if occ_start() then
            result ⟵ op(args) ; ▷ occ_traverse_done was
                added inside op's code
            if occ_finish() then
                return result
            end
        end
        occ_restart() ; ▷ Op aborted
    end
```
Listing 1: Code of macro-transformed operation *op*.

add `occ_node_born`/`occ_node_dies` calls to the node constructor/destructor.

Mechanizing the transformation. The transformation can be performed automatically by a source-to-source transformer tool, which we are in the process of implementing. The transformer requires the user to supply the tree's sequential source code, the names of methods to be macro-transformed and structure(s) implementing nodes, and to manually add the `occ_traverse_done` call. The transformer performs the following steps: ① Ensure that all node fields are accessed via setter/getter methods, by replacing every direct field access with an appropriate setter/getter call and generating getter/setter methods if they do not exist in the input code. ② Perform the micro-transformations by modifying methods in the node structure. ③ Generate the macro-transformed operations.

4.2 LCOW synchronization library

Occualizer's synchronization library has two high level tasks. First, it tracks the tree as observed by the operation, so that once the operation's traversal finishes, Occualizer can guarantee that the tree is in a consistent state from the operation's perspective. Second, the library buffers the operation's writes and exposes them atomically when the operation completes. We now walk through the library's flow.

Initialization (occ_start). This method checkpoints the thread's local state, so that execution can restart if the operation subsequently aborts. It then initializes the library's thread-local bookkeeping variables (Table 2), which track edges observed by the operation, nodes allocated, destroyed, locked, and copied, and other flags, such as whether the operation is in the midst of its traversal. We treat these variables as abstract datatypes for now (in particular, without considering implementation efficiency); § 5 describes our implementation.

Node reads (occ_get). We first focus on the case of reading a node pointer (child), i.e., reading an edge. As long as an operation is traversing the tree, its reads are handled with minimal overhead. The library only stores traversed edges in the `edgeSet`, to verify their consistency in case the operation rereads them in its post-traversal steps. Once the traversal

Name	Type	Content
`edgeSet`	Set of edges	Edges observed during operation
`lockedSet`	Set of nodes	Nodes locked by operation
`copySet`	Set of node pairs	Copied nodes and their copies
`bornSet`	Set of nodes	Nodes allocated by operation
`destroyedSet`	Set of nodes	Nodes destroyed by operation
`traversing`	Boolean	Initially *True*; *False* after `occ_traverse_done` called

Table 2: Thread-local bookkeeping structures. The term "node" refers to a pointer to the relevant object in memory.

completes, the library switches to a mode that guarantees consistency of the observed tree. This is accomplished by locking any node accessed post-traversal, while verifying that the locked node belongs to the most updated tree in memory. If this verification fails, the operation is aborted (releasing any locks it acquired and nodes it allocated) and restarted.

Listing 2 shows the pseudo code of `occ_get`. Suppose operation *op* is trying to read the *c*-th child of node *n*. (For generality, we assume child pointers are stored in a `children` vector in the node.) If *op* previously created a copy of *n*, the read is satisfied from the copy n' without further checks. Otherwise, *n*'s *c*-th child, *u*, is read from *n*. If *op* is traversing, the method *trackEdge* only remembers the edge (n,c,u) in `edgeSet`. Otherwise, *trackEdge* locks *n* by calling *lockNode*.

The *lockNode* method locks *n* while checking its consistency with *op*'s observations of the tree. If *n* is not present in *op*'s `lockedSet`, *op* tries to acquire *n*'s lock. If *n* is locked, *op* aborts, to avoid deadlocks. Then *op* checks that any edge into or out of *n* that *op* previously observed still exists. If so, *n* is added to *op*'s set of locked nodes. Otherwise, *op* aborts.

Reads of non-pointer fields are handled identically (locking the node, etc.) except that the read values are not tracked.

Node writes (occ_set). On any write to a node *n*, the library locks *n* and creates a copy of it, n'. If the operation completes successfully, n' will take the place of *n* in the tree and *n* will remain locked and hence immutable until its memory is reclaimed. Listing 3 shows the pseudo code of `occ_set`, again focusing on the case of writing a child pointer. If *op* has made a copy, v', of the pointed-to node *v*, the value to be written is changed to v'. Next, *op* checks if the written node *n* is part of the tree, i.e., it was not allocated by *op* itself and is not a copy. If so, *op* creates a copy of *n* by calling *occ_create_copy*. Finally, the write is performed.

To copy *n*, the *occ_create_copy* methods locks *n* using the *lockNode* method described above. It then copies the (now locked) *n* into a new node, n', records that *n* has a copy n' in *op*'s `copySet`, and finally adjusts the links between the existing copied nodes to reflect the new copy. For every $(x,x') \in$ `copySet`, the *fixLinks* method changes any edge

```
Function occ_get_child(n, c):
    if (n,n') ∈ copySet then
        return n'.children[c]
    else
        u ← n.children[c]
        trackEdge(n, c, u)
        return u
    end

Function lockNode(n):
    if n ∈ lockedSet then
        return
    if tryLock(n.lock) fails then
        abort operation
    ▷ Validate
    foreach (x,c,y) ∈ edgeSet, s.t. x = n or y = n do
        if x.children[c] ≠ y then
            abort operation
    end
    lockedSet.add(n)
```

Listing 2: Code of reading a node child..

```
Function occ_set_child(n, c, v):
    if (v,v') ∈ copySet then
        v ← v'
    end
    if n ∉ bornSet and (n,_) ∉ copySet then
        n ← occ_create_copy(n)
    end
    n.children[c] ← v

Function occ_create_copy(n):
    lockNode(n)
    n' ← copy of n
    copySet.add(n,n')
    fixLinks(n,n',copySet,bornSet)
    return copy
```

Listing 3: Code of writing a node child.

pointing from x' to *n* to point to n' instead, and changes any edge from n' pointing to *x* to point to x' instead. It similarly fixes any edge pointing to *n* from nodes allocated by *op*.

Writes of non-pointer fields are handled identically, except that value written is not "translated" as it is not a node pointer.

Traversal completion (occ_traverse_done). On traversal completion, the library switches to its post-traversal mode, in which any accessed nodes is locked. In addition, the last node read by the traversal is locked using the *lockNode* method. This locking is needed to guarantee that concurrent tree modifications do not "invalidate" the node's traversal (see § 6).

Node allocation/deallocation. On node allocation, the new node *n* is added to `bornSet`. On node destruction, the destroyed node *n* is locked (using *lockNode*) and added to `destroyedSet`. This is done so that *n* can be left immutable when the operation completes.

Operation commit (occ_finish). This method *commits* an operation by atomically applying its writes to the tree. Consider first the simple case in which the operation *op* finishes its traversal at node n and subsequently accesses nodes only down the tree. In this case, the paths to nodes written by *op* form a tree T_n rooted at n, and all nodes in T_n are locked by *op*. Occualizer can thus atomically apply *op*'s writes by creating a copy of T_n, T'_n, which contains the updated versions of the nodes *op* wrote to and then swinging the pointer to n from its parent, p, to point to n', the root of T'_n. (Figure 2, with p being the orange node.) We thus call T_n the *copied region*.

To ensure atomicity, *op* must verify that p is still in the tree and that the edge (p,c,n) that it swings has not changed since *op* originally crossed p during the traversal. If n is a non-root node, this is done by locking p and validating that $p.children[c] = n$ before overwriting it. If n is the root, this is done by locking a global root lock and validating the root's value (which also gets saved in `edgeSet`). Although we describe this parent validation step last, it chronologically occurs before copying any yet uncopied nodes from T_n, to avoid wasting CPU cycles in case *op* is doomed to abort.

Finally, nodes locked by *op* that are not in T_n are released. The nodes of T_n are *retired*, which means that their memory is freed/reclaimed once no concurrent operation can observe them. (Occualizer relies on an RCU-like epoch-based safe memory reclamation (SMR) management library [45, 72] to provide this functionality.) Until reclamation, these nodes remain locked and thus immutable.

In the general case, *op* may have proceeded up the tree, by accessing nodes it observed while traversing. The above discussion still holds, except that instead of taking n as the copied region's root, *occ_finish* needs to find the lowest common ancestor (LCA) of all written nodes and lock every path from that LCA to each written node. This is straightforward to do, because the LCA and all relevant edges have been read during *op*'s run (possibly only in the traversal step).

4.3 Optimizing range scans

An important feature of search trees is that they support *range scans*, the ability to iterate over the stored keys in order by using successor/predecessor calls. Specifically, we assume the tree implements a C++ standard library (STL)-like `iterator` object. The iterator maintains a key k', which is initially the predecessor or successor of its constructor argument. The iterator provides next/prev calls, each of which updates k' to its successor/predecessor, respectively. As with other concurrent search trees [9, 71], our goal is for the individual next/prev calls to be atomic (linearizable)—not for an entire range iteration performed by a sequence of such calls to be atomic with respect to insertions/deletions.

In sequential trees, a common method of implementing an iterator is to add *auxiliary* next/prev pointers to node fields, so that advancing an iterator does not require walking paths in the tree. Occualizer supports trees with such auxiliary edges, provided that they are symmetric (i.e., v points to u via an auxiliary link if and only if u points to v), as is the case of next/prev pointers. In addition, the user is required to specify the field names of auxiliary links in the node structure. Occualizer then leaves iterator movement over auxiliary edges as a read-only operation.

Extended commit protocol. We extend Occualizer's commit protocol to support auxiliary edges as follows. When an operation *op* is ready to commit, after having locked and validated p, the parent of r, the copied region's root, *op* checks each auxiliary edge (v,u) pointing to the copied region (i.e., such that u is in the copied region and v is not) and attempts to lock v (aborting if it fails).[4] Once all these "border" nodes are locked, *op* updates p to point from r to its new version r' with one atomic write, and then iterates over each locked border node v, updating its relevant auxiliary edges to point to the new version of the neighbor u (from (v,u) to (v,u')), and releasing v's lock afterwards. This protocol may seem heavyweight, but in practice copied regions tend to be small, so the extra cost of handling auxiliary edges is not substantial.

Unfortunately, the extended commit protocol breaks Occualizer's LCOW technique of replacing a copied region with one atomic memory write. The problem is that a sequential tree with auxiliary edges does not meet our PR1, because the auxiliary edges create more than one path to a node, and the commit protocol needs to update these paths when replacing a copied region. For example, in an external binary tree whose leaves are connected with next/prev links, Occualizer needs three writes to replace a copied region—to the parent of the region's root and to the predecessor and successor nodes of the region's leftmost and rightmost leaves, respectively.

Because Occualizer cannot *physically* atomically update all edges crossing the border between a copied region and the rest of the tree, our solution is to make the update *logically* atomic, as detailed below.

Logically atomic updates. To make iterators observe updates atomically, we ensure that an iterator only moves across auxiliary edges that exist in the latest version of the tree—i.e., edges that are not part of, or cross into, a copied region which is being replaced. To achieve this, Occualizer prevents iterators from moving to a locked node, relying on the fact that every node in a copied region is locked. When an iterator positioned at node v attempts to move to v's neighbor u and finds either v or u locked, the iterator instead "resynchronizes" its position using the latest version of the tree. Specifically, the iterator searches from the root for v's predecssor or successor x (as during iterator construction), according to where the iterator was trying to move. The iterator then positions itself at x and returns x's key.

This protocol guarantees that after an updating operation *op* exposes its writes (by updating some child pointer to link

[4] Our requirement that auxiliary edges are symmetric guarantees that *op* finds every node with an auxiliary edge to the copied region.

the new version of *op*'s copied region into the tree), no iterator can move into the copied region (whether the iterator is positioned inside or outside the copied region). Attempting such a move causes the iterator to "resynchronize" itself in the updated tree, which no longer contains the copied region. The protocol is conservative, however, in that while *op* holds the copied region locked but has not yet exposed its writes, an iterator moving across an auxiliary edge (v,u) to such a locked node u will "resynchronize" superfluously, ending up at u again, as it remains reachable from the root. Such a superfluous "resynchronization" does not violate the iteration's correctness; it can be thought of as reaffirming the iterator's position in the tree, with the iterator's move being linearized before *op*'s updates.

5 Implementation

In our Occualizer prototype, we perform the input code transformations manually (following the recipe of § 4.1) and implement the synchronization library in C++ using `pthread` spinlocks. This section describes the library's implementation.

Thread-local structures. We implement the various sets using thread-local dictionary (unordered map) objects, for efficient access. The `copySet` is implemented as a pair of maps, from nodes to their copies and vice versa. The `edgeSet` is implemented as a pair of maps, an *incoming* map that maps a node to its parent and child index there, and an *outgoing* map that maps a node to a list of its children indices read. The `lockedSet` is implemented as a map from locked nodes to a boolean indicating if the lock should be released on commit. The other sets are implemented as C++ STL vectors.

Efficient `copySet` searches. We further optimize `copySet` searches by adding a flag into the node structure which is set when a node is copied. The `occ_get` method uses this flag to avoid superfluous `copySet` searches. When an operation aborts, it clears this flag from all the nodes it copied.

Optimized dictionaries. Our initial implementation used C++ STL hash tables (`unordered_map`), but we observed that they impose considerable overhead. We therefore replace them with an optimized design, which initially inserts items into a small STL vector, and if the vector becomes full, stores overflowing items in an `unordered_map`. The observation underlying this optimization is that in most tree algorithms, the thread-local data structures Occualizer maintains will be small. But for correctness, we must support worst-case behavior in which these structures may contain every node in the tree. Overall, this optimization improved the throughput of an Occualizer B+tree by a factor of two.

Correctness testing. We use a couple of testing techniques to gain confidence in the correctness of the Occualizer prototype. First, we use the linearizability checking option of the SetBench [13] benchmarking harness (§ 7.1). With this option, SetBench verifies that every successful insert/delete operation during the execution is correctly reflected in the final state of the tree—so that, for example, inserted items were not lost or inserted more than once. We test different tree sizes, to test executions with varying contention levels and thread interleavings. Second, we check that tree-structural invariants of the sequential implementations we transform (e.g., the red-black property of a red-black tree) hold, both at random times during the execution and after it completes.

6 Correctness

This section sketches the proof that trees produced by Occualizer are linearizable [56], i.e., tree operations appear to execute atomically. We consider the shared-memory system running the tree. A *state* of the system consists of the memory state (contents of each address) and the local states of each thread. In each *step* of the execution, some thread accesses memory, and as a result, its internal state and/or memory change.

Our proof works as follows. We first show that traversals are correct. That is, if $traverse(k)$ stops at node v in the concurrent execution, then at some point during its run, the memory state σ was such that had $traverse(k)$ executed atomically (from start to finish) on σ, it would also reach v. We denote this property of a state σ by $\sigma :\overset{k}{\rightsquigarrow} v$. We then use traversal correctness and Occualizer's synchronization protocol to show that if an operation *op* commits in memory state σ, then had *op* run from start to finish on σ, it would have executed exactly the same. Hence, *op* appears to execute atomically at σ.

Showing traversal correctness is hard, because traversals are unsynchronized and can observe inconsistent tree states. We solve this problem by applying the theorem of Feldman et al. [44], which says that in a concurrent tree satisfying a "forepassed" condition, unsynchronized traversals are correct. The "forepassed" condition requires (1) traversals to be single-step, which we satisfy by PR3; and (2) that if the concurrent algorithm performs a write w moving the system from state σ to σ', such that $\sigma :\overset{k}{\rightsquigarrow} v$ but $\sigma' :\overset{k}{\not\rightsquigarrow} v$ for some k and v, then v is never modified later.

Requirement (2) above follows from PR5 and the fact that Occualizer leaves written/destroyed nodes immutable (locked). There is a subtle issue, however, which is that the prerequisites are met by the sequential code, so unless we know operations in the Occualizer tree behave correctly, we cannot rely on the prerequisites. But we need traversal correctness to prove this fact, creating a "chicken and egg" problem.

We address this problem using a proof technique suggested by Feldman et al. for proving "forepassed" is satisfied [44, §7]. The technique is to prove both that "forepassed" is satisfied and that the concurrent tree is correct in tandem, inductively (on steps of the execution), so that each proof can rely on the other property holding on the execution thus far.

Accordingly, we prove correctness of an Occualizer tree assuming traversal correctness. The proof is inductive: we need to show that in every state σ, if $op(k)$ runs sequentially on σ, its post-traversal memory accesses are identical to its post-traversal memory accesses so far.

In the base case, σ is when *op*'s traversal's stops at node v. Traversal correctness implies that for some σ' during *op*'s execution, $\sigma' :\overset{k}{\rightsquigarrow} v$. Now, Occualizer locks v. Based on the induction hypothesis, a lock acquisition implies that $\sigma :\overset{k}{\rightsquigarrow} v$—i.e., v is on the search path for k "now"—as otherwise, v would have been locked and copied between σ' and σ and *op*'s lock acquisition would have failed. Thus, if *op* runs sequentially at σ, its traversal would reach v. But the lock acquisition then implies that the traversal would also *stop* at v, since only pointer fields in v could have changed, but *op* verifies that any pointer it read did not change after locking v.

The inductive step is similar. We are in state σ with *op* successfully locking some node u. Inductively, we know that (1) in some σ' in the past, the state of the tree was such that *op*'s execution on it would have lead it to its current local state, and (2) any modifications made to the tree since σ' have only moved it through consistent states. Moreover, due to *op*'s locking, these changes in tree state do not change the nodes and edges *op* has observed post-traversal. It follows from PR4 that if we run *op* in state σ, it will behave exactly the same. This concludes the overall proof, when σ is the state in which *op* commits its writes.

7 Evaluation

We compare Occualizer trees to mechanically-crafted trees (§ 2) and to hand-crafted trees, with respect to scalability, throughput, and memory use. We first evaluate the trees on workloads with different amounts of writing operations (§ 7.1). We then focus on an Occualizer B+tree: we compare it as an index in the STOv2 main-memory database to the default Masstree index (§ 7.2), and analyze its overhead (§ 7.3).

Transformed trees. We use Occualizer to create concurrent versions of the following sequential trees:

- **B+tree**: An improved version of the optimized STX in-memory B+tree [7] taken from the `tlx` library [8].
- **Radix**: An implementation of a radix tree (trie) [79]. The code follows the description of Linux's radix tree [24].
- **RB**: A red-black tree [32]. The code is the sequential implementation used in Synchrobench [47].

We refer to a transformed tree implementation T as *occ[T]*.

Experimental platform. We use a dual-node NUMA server. Each node has a 14-core Intel Xeon Gold 6132 (Skylake) processor and 96 GB of DDR4-2666 DRAM. Hyper-Threading and Turbo-Boost are disabled. Threads are split between the nodes and memory allocation is interleaved across the nodes. Code is compiled using GCC 8.3.0 and linked with the `jemalloc` [37] multi-threaded memory allocator. Reported numbers are averages of 10 runs; all measurements are within $\pm 5\%$ of the average.

7.1 Contention benchmarks

We compare Occualizer's trees to mechanically- and hand-crafted trees on workloads with increasing amounts of writing operations.

Trees. We compare to the following trees, which unless noted otherwise are mechanically-crafted from the same sequential code used for Occualizer:

- **Global-Lock**: Created by serializing operations with a global lock.
- **GCC-TM** [1]: Created by wrapping operations in transactions using GCC's transactional memory (TM) support. The underlying TM algorithm uses optimistic concurrency control with eager locking and tracks conflicts at word granularity.
- **CX** [26]: Created with the CX universal construction, which produces wait-free operations and does not serialize read-only operations. It is the fastest wait-free universal construction we are aware of, although it still copies the entire data structure. We use the original authors' implementation [27].
- **COW** [5]: Created with a COW-based approach inspired by Ben-David et al., which produces lock-free writing operations and wait-free read-only operations, without serializing read-only operations. We implement COW ourselves.
- **Hand-Crafted:** We use hand-crafted designs of comparable algorithms, as we are not aware of concurrent implementations of exactly the same trees. We compare occ[RB] to SnapTree, a concurrent AVL tree with optimistic (lock-based) synchronization [11]. We compare occ[Radix] to a lock-free version from the same repository [79]. We compare occ[B+tree] to Brown's lock-free B-slack tree [12] (a B-tree variant). We use the original authors' implementations.

Our tree selection covers a spectrum of mechanically- and hand-crafted synchronization techniques. While some of these techniques do not form an "apples to apples" comparison with Occualizer's optimistic lock-based synchronization (e.g., due to being lock- or wait-free), the point is that they represent the space of currently available mechanical techniques.

Workloads. We populate the tree with 64 M uniformly random 8-byte keys and then run a workload for 3 seconds. Each workload has a different mix of operation types (Table 3). Our workloads are inspired by the standard Yahoo! Cloud Serving Benchmark (YCSB) [23] workloads, which are designed to simulate real-world application workloads, but differ in that (1) we replace updates with insertions and do not test range

Workload	Description
R-100	100% lookups (YCSB-C)
R-97	97% lookups, 3% insertions (≈ YCSB-B)
R-75	75% lookups, 25% insertions
R-50	50% lookups, 50% insertions (≈ YCSB-A)

Table 3: Contention workloads.

queries, as not all implementations support these operations; and (2) we use more levels of writing operations, for more fine-grained insight. Experiments run under SetBench [13], a benchmarking harness for concurrent C++ dictionaries that provides epoch-based memory reclamation.

Throughput. Figures 4–6 show the aggregate throughput for each workload with varying numbers of threads for the B+tree, radix, and red-black tree variants, respectively. All variants except Global-Lock scale on the read-only workload (R-100), although GCC-TM's throughput drops for RB at 24–28 cores. But for workloads with any level of writing operations, only Occualizer and the hand-crafted trees scale, with throughput of mechanically-crafted trees typically flatlining at low core counts.[5]

The mechanically-crafted trees do not scale due to suboptimal synchronization. In CX and COW, all writing operations are serialized, although serialization in COW, which is done with a CAS to the tree root, is significantly faster than in CX, where writing operation participate in a helping scheme and may copy the entire data structure. In GCC-TM, the problem is its conservative version of optimistic synchronization, which retries any transaction if any node it reads is updated before the transaction commits. Thus, for example, GCC-TM breaks down on RB—where tree rebalancing writes to the top of the tree—much earlier than on Radix.

Compared to the hand-crafted trees, Occualizer achieves comparable or better throughput when the level of mutation is low (R100-R97). The cases in which Occualizer is faster are due to differences in the underlying algorithms, which manifest when synchronization overhead is low. This effect demonstrates the power of Occualizer's approach, which removes the difficulty of adding synchronization to a tree from consideration, and thereby allows focusing on the (sequential) "quality" of the tree, i.e., how fast it is to search.

As mutations increase (R75-R50), however, synchronization becomes the dominating factor and Occualizer significantly underperforms the hand-crafted trees. The reason is that Occualizer writing operations are slower than in the hand-crafted trees, due to bookkeeping and copying overhead (see § 7.3), and so as the proportion of mutations grows, overall throughput degrades.

[5]The only exception is GCC-TM on Radix. In Radix, the tree only grows downwards, so any non-null pointer read during a traversal is immutable. GCC-TM thus rarely aborts transactions even with moderate mutation levels, and so achieves high throughput.

Tree	Throughput relative to hand-crafted	R-100	R-97	R-75	R-50
B+tree	Occualizer	0.79	0.77	0.64	0.72
	Best mech-crafted	0.82	0.38	0.05	0.05
	Gap shrink	—	**2.01**	**12.28**	**13.54**
Radix	Occualizer	1.08	0.84	0.73	0.50
	Best mech-crafted	1.15	0.90	0.74	0.36
	Gap shrink	—	**0.93**	**0.99**	**1.38**
RB	Occualizer	1.01	0.90	0.31	0.19
	Best mech-crafted	1.02	0.24	0.04	0.03
	Gap shrink	—	**3.78**	**8.64**	**6.08**

Table 4: Throughput difference between Occualizer and the best result of the mechanically-crafted trees at 28 cores, for each workload.

Tree	B+tree	Radix	RB
occ[·]	2.14 GB	2.10 GB	3.12 GB
Global-Lock	0.91×	0.93×	0.93×
GCC-TM	0.91×	0.96×	0.95×
CX	13.5×	13.18×	13.03×
COW	1.02×	1.18×	1.04×
Hand-Crafted	1.12×	0.97×	1.01×

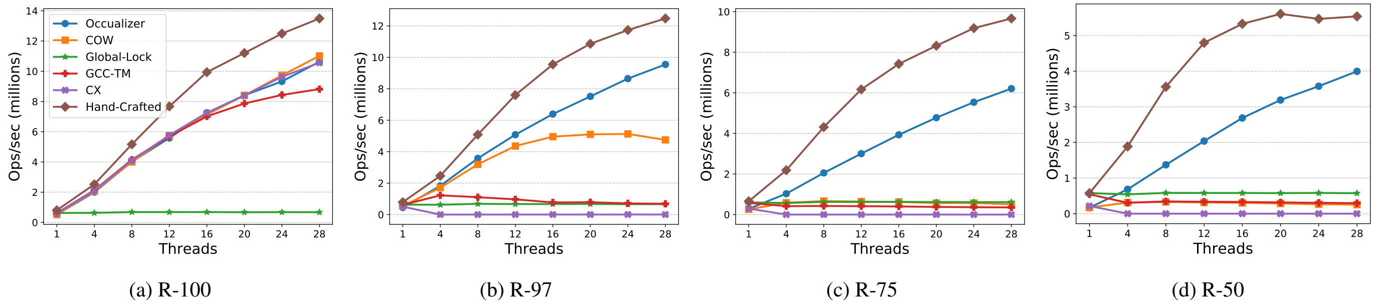
Table 5: Memory use at 28 cores (R-50), normalized to Occualizer's.

The takeaway is that Occualizer significantly shrinks the performance gap between mechanically- and hand-crafted trees in workloads with mutations. Table 4 reports this gap, and by how much Occualizer shrinks it. Overall, Occualizer shrinks the gap by up to 13.54×, 1.38×, and 8.64× for B+tree, Radix, and RB, respectively.

Memory use. Node copies made by an Occualizer tree may increase memory use compared to its sequential version, as a function of how quickly old nodes are reclaimed. To quantify this effect, Table 5 shows peak memory use for each tree in the R-50 workload (results for other workloads are similar). Both Occualizer and COW indeed use more memory than the sequential baseline (captured by Global-Lock), but Occualizer's LCOW increases memory use by 7%–9% whereas COW, which copies entire paths, adds an overhead of 11%–26%. In contrast, CX uses about 14× the memory of Global-Lock. The reason is that CX maintains multiple replicas of the data structure, so that read-only operations can read from a replica and avoid being serialized. Results of the hand-crafted algorithms are shown only for completeness, as they are not implementations of the same algorithm.

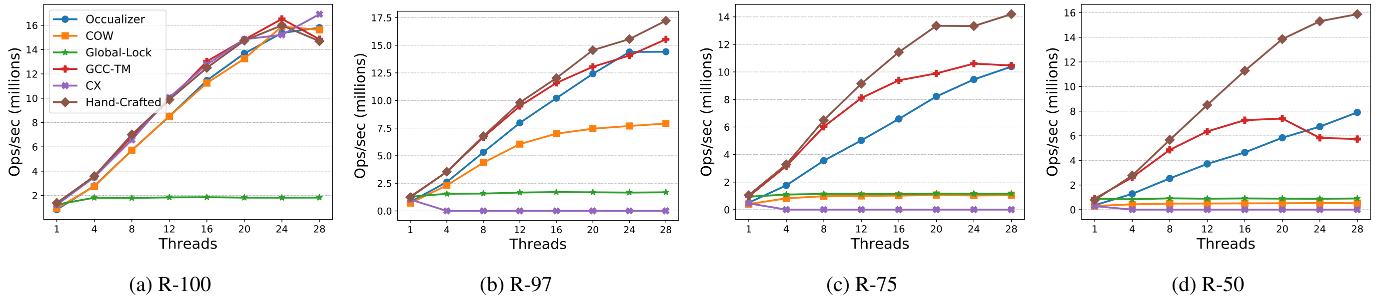
7.2 Full-system benchmark

We add occ[B+tree] as the index data structure in the STOv2 main-memory database system [58], and compare the result to the default index, Masstree [71], a hand-crafted concurrent



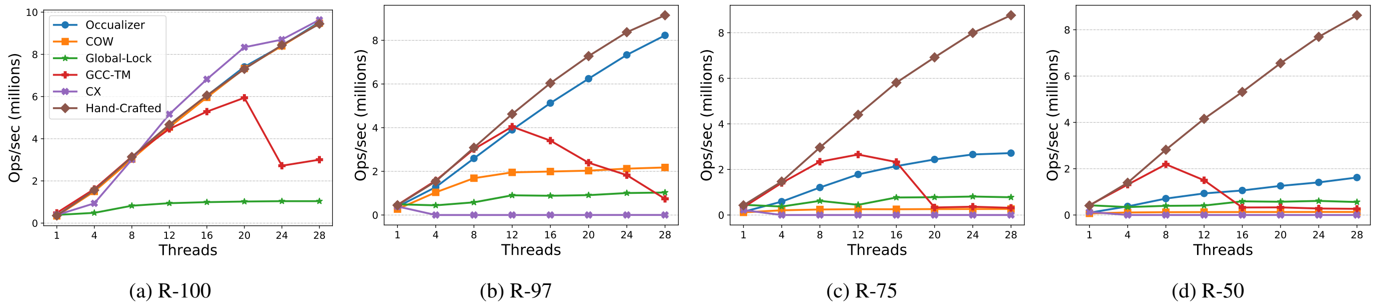
(a) R-100 (b) R-97 (c) R-75 (d) R-50

Figure 4: Throughput of B+tree variants for workloads with increasing amounts of mutation.



(a) R-100 (b) R-97 (c) R-75 (d) R-50

Figure 5: Throughput of radix tree variants for workloads with increasing amounts of mutation.



(a) R-100 (b) R-97 (c) R-75 (d) R-50

Figure 6: Throughput of RB (red-black tree) variants for workloads with increasing amounts of mutation.

tree combining aspects of B+trees and tries.[6]

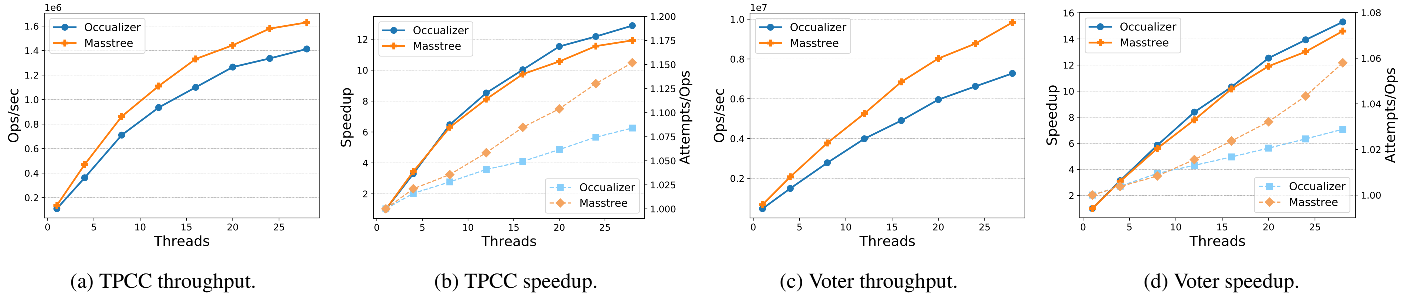
STO provides serializable transactions. Transactions are specified via C++ programs accessing STO's transactional interface. STO's transaction concurrency control is tightly coupled with Masstree, as it relies on Masstree node version numbers to detect and block certain anomalies. To integrate occ[B+tree] into STO, we therefore implement an equivalent versioning scheme in the `tlx` B+tree.

Workloads. We evaluate two transactional workloads, TPC-C and Voter. TPC-C is the industry standard benchmark for evaluating the performance of online transaction processing (OLTP) systems [28], by simulating an order processing application. We use a database with one warehouse, 100,000 items, and run the full mix of all TPC-C transactions. This workload performs index range queries. Voter is a benchmark that simulates a phone-based voting application. It consists of many short transactions and does not perform range queries.

Results. Figure 7 shows the throughput (committed transactions per second) and scalability of the system for both workloads, measured over a 10-second run. Scalability is measured by normalizing the throughput obtained with each index to the single-threaded throughput obtained with that index. On both workloads, occ[B+tree] is slower than Masstree, but has better scalability. As a result, the performance gaps between them shrinks as more threads are added: from a single-threaded difference of 22% and 29% for TPC-C and Voter, respectively, to a difference of 12% and 26% at 28 threads.

The reason behind occ[B+tree]'s better scalability is that Occualizer's optimistic synchronization protocol causes fewer operations to abort and retry than Masstree's protocol (which is also optimistic). Masstree uses per-node version counters to guarantee that searches observe only consistent node states. In Masstree's protocol, any operation—including a lookup—might abort and retry if its version checking indicates it may

[6] We use a B+tree here because it is closest algorithmically to Masstree, which is a B+tree variant. Comparing to, say, a red-black tree would not be meaningful, because Masstree outperforms a red-black tree for reasons unrelated to concurrency control (e.g., its much "shallower" tree structure).



(a) TPCC throughput. (b) TPCC speedup. (c) Voter throughput. (d) Voter speedup.

Figure 7: Throughput and speedup (throughput normalized to single-threaded throughput) of occ[B+tree] vs. Masstree in STOv2. Speedup plots also show the average number of attempts required to complete an operation (right Y axis).

Enabled methods	Relative throughput
None (call overhead)	93%
`occ_start`	87%
+ `occ_finish`	81%
+ `occ_get`	53%
+ `occ_set` = occ[B+tree]	40%

Table 6: Single-threaded throughput of occ[B+tree] relative to sequential B+tree (R-50 workload) as Occualizer's synchronization functionality is gradually enabled.

have observed an inconsistent node state. In contrast, Occualizer relies on LCOW to guarantee that observed nodes are consistent, and on the "forepassed" condition [44] to guarantee correctness of searches that traverse inconsistent tree states. In Occualizer's protocol, read-only lookups never abort and retry—only update operations do. As a result, as Figures 7b and 7d show (on the right Y axis), the average number of attempts required to complete an operation is larger in Masstree than in occ[B+tree]—and the difference grows with the number of threads.

7.3 Overhead analysis

To break down the sources of Occualizer's performance overhead, we evaluate the throughput impact of making the transformation but using a no-op implementation of each library method, then gradually adding in each method's actual implementation. We use single-threaded execution for this evaluation, because an Occualizer tree does not run correctly when any of the methods are disabled.

Table 6 shows the results, comparing occ[B+tree] to the original sequential B+tree, on the R-50 workload. The lion's share of overhead is due to `occ_get` and `occ_set`, which interpose on node field accesses. The impact of `occ_get` is $\approx 2\times$ that of `occ_get`, as reading fields is more frequent. Invoking the methods, `occ_start`, and `occ_finish` each degrade throughput by 6–7% points.

8 Conclusion

This paper presented Occualizer, a mechanical transformation for adding scalable optimistic synchronization to a sequential search tree implementation. Occualizer's specialization to trees enables designing a synchronization protocol that does not suffer from the limitations of transactional memory and universal constructions. Overall, Occualizer trees shrink the performance gap between these automatic transformations and hand-crafted trees by up to $13\times$.

Occualizer is limited, however, in that it applies only to sequential search trees that satisfy its prerequisites. Relaxing the prerequisites and automating the verification that an input tree satisfies them are interesting future directions, as is reducing the overhead of Occualizer's synchronization library. Our current Occualizer prototype also requires some manual steps to transform the input tree; automating these steps is an ongoing effort.

Occualizer's code is available at https://github.com/tomershanny/Occualizer.

Acknowledgements

We thank Yotam Feldman for many illuminating and enjoyable (often simultaneously) discussions. We thank the reviewers and the paper's shepherd, Irina Calciu, for their feedback.

This research was funded in part by the Israel Science Foundation (grant 2005/17) and the Blavatnik Family Foundation.

References

[1] The GNU Transactional Memory Library. https://gcc.gnu.org/onlinedocs/gcc-5.5.0/libitm.pdf, 2021.

[2] James H. Anderson and Mark Moir. Universal Constructions for Multi-Object Operations. In *PODC*, 1995.

[3] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE TPDS*, 10(12), 1999.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[5] Naama Ben-David, Guy E. Blelloch, Yihan Sun, and Yuanhao Wei. Multiversion Concurrency with Bounded Delay and Precise Garbage Collection. In *SPAA*, 2019.

[6] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[7] Timo Bingmann. STX B+ Tree C++ Template Classes. https://panthema.net/2007/stx-btree, 2013.

[8] Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers. https://panthema.net/tlx, 2018.

[9] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD '18*, 2018.

[10] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT, 2014.

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. In *PPoPP*, 2010.

[12] Trevor Brown. B-slack Trees: Space Efficient B-Trees. In *SWAT*, 2014.

[13] Trevor Brown. SetBench: Powerful tools for data structure experiments in C++. https://gitlab.com/trbot86/setbench, 2021.

[14] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *OPODIS*, 2013.

[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *ASPLOS*, 2017.

[16] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*, 2001.

[17] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*, 2018.

[18] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A Universal Construction for Wait-Free Transaction Friendly Data Structures. In *SPAA*, 2010.

[19] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, 2012.

[20] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *EuroSys*, 2013.

[21] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP*, 2013.

[22] Douglas Comer. Ubiquitous B-Tree. *ACM CSUR*, 11(2), 1979.

[23] Brian F. Cooper, Adam Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*, 2010.

[24] Jonathan Corbet. Trees I: Radix trees. https://lwn.net/Articles/175432/, 2006.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. The MIT Press, 2022.

[26] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A Wait-Free Universal Construct for Large Objects. In *PPoPP*, 2020.

[27] Andreia Correia, Pedro Ramalhete, and Pascal Felber. CX source code. https://github.com/pramalhe/CX, 2020.

[28] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.

[29] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, 2015.

[30] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[31] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *DISC*, 2006.

[32] Dave Dice, Nir Shavit, and Ori Shalev. Red-black balanced binary search tree. https://github.com/gramoli/synchrobench/blob/master/c-cpp/src/trees/rbtree/rbtree.c, 2006.

[33] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *ICAC*, 2014.

[34] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *PACT*, 2014.

[35] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal Constructions That Ensure Disjoint-Access Parallelism and Wait-Freedom. In *PODC*, 2012.

[36] Rob Ennals. Software transactional memory should not be obstruction free. Technical Report IRC-TR-06-052, Intel Research, 2006.

[37] Jason Evans. Scalable memory allocation using jemalloc. http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919, 2011.

[38] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *DISC*, 2009.

[39] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *PPoPP*, 2012.

[40] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory of Computing Systems*, 55(3), 2014.

[41] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[42] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic Transactions. In *DISC*, 2009.

[43] Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of Chaos: Proving Linearizability Using Local Views. In *DISC 2018*, 2018.

[44] Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. Proving Highly-Concurrent Traversals Correct. *PACMPL*, 4(OOPSLA), 2020.

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

[46] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-Structured Data Stores. In *EuroSys*, 2015.

[47] Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, 2015.

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

[49] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic Transactional Boosting. In *PPoPP*, 2014.

[50] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *SPAA*, 2010.

[51] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13, 1991.

[52] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *SIGOPS OSR*, 26(2), 1992.

[53] Maurice Herlihy and Eric Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *PPoPP*, 2008.

[54] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.

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

[56] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12, 1990.

[57] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-Aware Transactions for Faster Concurrent Code. In *EuroSys*, 2016.

[58] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. In *VLDB*, 2020.

[59] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline Index: A Unified Data Structure for Processing Queries on Temporal Data in SAP HANA. In *SIGMOD*, 2013.

[60] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[61] Siddharth Krishna, Nisarg Patel, Dennis Shasha, , and Thomas Wies. Verifying Concurrent Search Structure Templates. In *PLDI*, 2020.

[62] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL*, 2(POPL), 2018.

[63] Michael Larabel. Intel To Disable TSX By Default On More CPUs With New Microcode. https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode, 2021.

[64] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), 2006.

[65] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE '13*, 2013.

[66] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A Constructive Approach for Proving Data Structures' Linearizability. In *DISC 205*, 2015.

[67] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-Tree for New Hardware Platforms. In *ICDE '13*, 2013.

[68] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*, 2011.

[69] Linux. lib/rbtree.c, source code file of Linux 5.17. https://github.com/torvalds/linux/blob/v5.17/lib/rbtree.c#L35-L57, 2022.

[70] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *USENIX ATC*, 2012.

[71] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*, 2012.

[72] Paul E. McKenney and John D. Slingwine. Read-copy update: using execution history to solve concurrency problems. In *PDCS*, 1998.

[73] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, 2010.

[74] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *PDSIA*, 1999.

[75] Darko Petrović, Thomas Ropars, and André Schiper. On the Performance of Delegation over Cache-Coherent Shared Memory. In *ICDCN*, 2015.

[76] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing Conflicting Transactions in an STM. In *PPoPP*, 2009.

[77] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, 1995.

[78] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *JACM*, 32, 1985.

[79] Hugo Sousa. RadixTree. https://github.com/ha2398/radix-tree, 2016.

[80] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.

[81] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[82] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.

[83] Ziqi Wang, A. Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, M. Kaminsky, and D. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD '18*, 2018.

[84] Lingxiang Xiang and Michael Lee Scott. Compiler Aided Manual Speculation for High Performance Concurrent Data Structures. In *PPoPP*, 2013.

[85] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*, 2016.