Jiffy: A Lock-free Skip List with Batch Updates and Snapshots

Tadeusz Kobus
Poznan University of Technology
Poznań, Poland
tadeusz.kobus@cs.put.edu.pl

Maciej Kokociński
Poznan University of Technology
Poznań, Poland
maciej.kokocinski@cs.put.edu.pl

Paweł T. Wojciechowski
Poznan University of Technology
Poznań, Poland
pawel.t.wojciechowski@cs.put.edu.pl

Abstract

In this paper we introduce Jiffy, the first lock-free, linearizable, ordered key-value index that offers both (1) batch updates, i.e., put and remove operations that are executed atomically, and (2) consistent snapshots used by, e.g., range scan operations. Jiffy is built as a multiversioned lock-free skip list and relies on system-provided timestamps (e.g., on x86_64 obtained through the Time Stamp Counter register) to generate version numbers at minimal cost. For faster skip list traversals and better utilization of CPU caches, key-value entries are grouped into immutable objects called *revisions*. By (automatically) controlling the size of new revisions, our index can adapt to varying contention levels (e.g., smaller revisions are more suited for write-heavy workloads). Structure modifications to the index, which result in changing the size of revisions, happen through (lock-free) skip list node split and merge operations that are carefully coordinated with the update operations. Despite rich semantics, Jiffy offers highly scalable performance across varied workloads. Compared to Jiffy's lock-based rivals that support batch updates, our index can execute large batch updates up to 7.4 times more efficiently. Moreover, Jiffy often outperforms the state-of-the-art lock-free ordered indices that feature linearizable range scan operations but lack batch updates.

CCS Concepts • **Theory of computation** → **Concurrent algorithms**; **Data structures design and analysis**.

Keywords ordered index, lock-free skip list, batch update, snapshot, linearizability

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508437

1 Introduction

Concurrent programming is notoriously difficult. Hence, to develop applications and complex systems, such as database engines, which are optimized for modern multicore hardware, programmers often rely on *concurrent data structures*. These structures expose a well defined interface and can be safely used in a multithreaded environment without additional synchronization (see, e.g., [26]). Under the hood, concurrent data structures feature sophisticated, often non-blocking synchronization algorithms optimized for performance. With the proliferation of multicore hardware in recent years, many new concurrent data structures have been proposed, e.g., concurrent lists [35, 59], sets [18, 31, 38, 47, 54, 56], (ordered) key-value indices (or maps, dictionaries) [15, 16, 19–21, 32, 33, 46, 49, 52, 53, 55, 57, 58, 60], etc., each time improving the performance over the existing solutions and introducing new features, e.g., the support for consistent range scan operations or snapshots that provide a read-only, static and consistent view over the state of the entire dataset.

In this paper, we introduce *Jiffy*, the first linearizable [37], lock-free ordered index (sorted key-value map) that besides offering consistent snapshots used, e.g., by range scans, provides support for *batch updates*, i.e., put and remove operations that are executed atomically. The latter feature is often demanded by programmers: batch updates are part of API of Google's LevelDB [7] and Facebook's RocksDB [11], and are extensively used in large open-source projects (see, e.g., Apache Hadoop [1], Apache Spark [2] or Bitcoin [3]). However, providing batch updates in an in-memory index, which must be much faster than its SSD-backed counterparts, poses new challenges related to concurrent synchronization. We propose several innovations to make our in-memory index highly scalable, despite the rich semantics it offers. The novelty of our approach is characterized below and in Section 2.

The design of our index is based on a multiversioned [17] skip list [50]. Unlike existing multiversioned concurrent indices which rely on a single atomic counter to generate version numbers [16, 41, 42], Jiffy's concurrency control mechanism is designed to use system-provided timestamps as version numbers. For example, on the x86_64 platform, timestamps can be obtained through CPU's Time Stamp Counter (TSC) register [39, 51], a cycle-level resolution clock, whose value can be read without performing a system call. In turn, the versioning mechanism in Jiffy does not feature a single

point of contention and offers scalable performance for read *and* update operations on modern 40+ core CPUs.

Our unique versioning mechanism allows Jiffy to provide programmers with more robust snapshots compared to the existing lock-free indices. For example, in KiWi [16] or LFCA tree [60], snapshots are created and managed only by range scans. Consequently, with these indices a programmer *cannot* explicitly acquire a snapshot and freely read any keys or perform repeated range scans (as one could do using a read-only transaction in DBMS). Moreover, in LFCA tree the cost of creating a snapshot (for a range scan) is $O(n)$, where n is the number of entries returned by the range scan.

In Jiffy, all range scans are performed on consistent snapshots. However, one can also explicitly acquire a snapshot of the current state of the dataset and perform on it any combination of read-only operations: lookups (gets), ascending/descending range scans, relational operations (higherKey, lowerKey), acquire submaps, etc. A snapshot acquired explicitly can be easily refreshed to reflect a more recent state of the index, and eventually has to be disposed of (closed). The cost of acquiring/refreshing any snapshot is $O(1)$.

Key-value entries are grouped in Jiffy into immutable objects, called *revisions*, which are tagged with a version number. This way the index can be smaller and thus quicker to traverse. Moreover, accesses to individual key-value entries can be performed more efficiently through the use of a light-weight hash index inside each revision, whereas range scans can benefit from keys and values being stored in sorted arrays within the revision. Crucially, however, by growing or shrinking the skip list and thus modifying the sizes of revisions, we can optimize the synchronization granularity in Jiffy, which allows it to adapt to changing workloads. Smaller revisions are more suited for write-heavy workloads whereas large revisions benefit read-dominated workloads, especially when they feature many range scan operations. Automatic adaptation to the workload is accomplished on per-revision basis by monitoring the time threads spend executing updates and reads, not by counting the number of operations performed or monitoring the contention on shared references, as in other existing approaches, e.g., [52, 53, 60].

The core contribution of our paper is, however, the novel lock-free algorithm that enables updates, reads, and index structure modifications, which drive the changes in revision sizes. Structure modifications in Jiffy are very different compared to a classic skip list, as they involve splits and merges of nodes on the lowest level of the skip list. Node splits and merges are solely based on the atomic compare-and-swap (CAS) operations and are streamlined with updates (including batch updates) for minimal overhead.

The test results show that Jiffy (which we implemented in Java, see also [5]) exhibits scalable behaviour under varied workloads. Compared to its lock-based rivals that offer both batch updates and linearizable range scans [60], Jiffy can execute large batch updates much more efficiently thanks to its lock-free architecture, with speedup in throughput ranging from 1.1× to 7.4×. Moreover, despite more costly updates (due to the support of batch updates), Jiffy often outperforms the state-of-the-art lock-free ordered indices that lack support for batch updates [19–21, 53]. Jiffy is roughly 2× faster than the ubiquitous `ConcurrentSkipListMap` [27], which does not support batch updates nor linearizable range scans.

To test and debug Jiffy, we developed a test harness for concurrent data structures. During our test, concurrent threads log traces of performed operations (the traces also include system-provided timestamps). Traces are then transformed into a graph that reflects various relationships between the logged events, e.g., write-read/write-write dependencies, program and timestamp order, batch update and snapshot-based constraints, etc. We iteratively refine the graph by inferring new relationships between events and check whether the graph is still acyclic. The test harness, which is part of Jiffy's codebase, gives us real confidence in our implementation, despite its complexity.

1.1 Paper structure

The paper has the following structure. In Section 2 we discuss work closely related to ours. Then, in Section 3, we describe the design of our system, including the system architecture, the way we implemented all operations (put, remove, batch update, lookup, and range scans), additional internal operations (structure modifications), the internal data structures, and the used autoscaling policy. Then, we discussed the correctness of our system. In Section 4, we present the results of experimental evaluation. We conclude in Section 5.

2 Related work

A template for non-blocking concurrent data structures based on CAS was originally proposed in [36]. Implementations based on this approach suffer from low parallelism and high overhead due to excessive copying and reliance on a single global pointer accessed through CAS by all threads. Much better performing ordered indices rely on purposefully designed (non-blocking) algorithms (discussed below and summarized in Table 1).

Skip lists were first introduced by Pugh [50]. Valois [59] was the first to sketch a lock-free algorithm for a skip list, although the first complete algorithm was proposed by Sundell and Tsigas [58], as an extension of their prior work on concurrent priority queues [57]. Frasier [33] gave an alternative implementation of a lock-free skip list, which relies on Harris' CAS-based approach for implementing lock-free linked lists [35]. Fomitchev and Ruppert's implementation of a lock-free skip list [32] combines the techniques of Valois and Harris. The ubiquitous `ConcurrentSkipListMap` [27], which is part of `java.util.concurrent` library, draws from

Table 1. Summary of properties of selected ordered indices.

	Basic architecture	Concurrency scheme	Batch updates	Linearizable range scans	Linearizable snapshots
Java CSLM [26]	skip list	lock-free			
SnapTree [19]	AVL tree	locks		✓	✓
CTrie [49]	hash trie	lock-free		✓	✓
Minuet [55]	dist. MVCC B-tree	locks	✓	✓	✓
k-ary tree [20, 21]	k-ST	lock-free		✓	
LeapList [15]	skip list	locks + STM		✓	
CA-imm [52]	BST	locks		✓	
Nitro [41]	MVCC skip-list	lock-free		✓	✓*
KiWi [16]	MVCC skip list	lock-free		✓	
LFCA tree [60]	BST	lock-free		✓	
CA-AVL [53]	BST	locks	✓	✓	
CA-SL [53]	BST	locks	✓	✓	
Jiffy	MVCC skip list	lock-free	✓	✓	✓

* In Nitro, snapshot creation is not a thread-safe operation.

Freiser's, Fomitchev's and Sundell's work. All algorithms discussed above are linearizable [37] except for range scans, and, unlike Jiffy, they do not support batch updates or snapshots.

LeapList [15] and KiWi [16] are skip list-based indices that provide linearizable range scans (but no fully linearizable snapshots that can be acquired on demand by a programmer, as in Jiffy). LeapList relies on fine-grained locks and Software Transactional Memory for concurrency control. On the other hand, KiWi features a multiversioned architecture and CAS-based operations to provide lock-freedom (range scans are wait-free). In KiWi, not every update creates a new version: without concurrent range scans, an update operation simply overwrites the old value in the index. Version numbers are managed using an atomic counter, which can be a bottleneck (in Jiffy we rely on system-provided timestamps instead). Each of the base nodes in LeapList and KiWi holds k key-value entries for cache-friendliness, but k is fixed (unlike in Jiffy).

Nitro [41], a skip list-based index used in Couchbase [4], uses multiversioning to provide snapshots, but the creation of a new snapshot is not a thread-safe operation (it cannot be executed concurrently with put/remove operations).

Now we discuss tree-based ordered index data structures. SnapTree by Bronson *et al.* [19] is a lock-based relaxed balance AVL tree. SnapTree uses a linearizable clone operation for atomic snapshots and range scans, which can severely slow down concurrent update operations. In Jiffy, creating a snapshot, which is also used for a range scan, is an O(1) operation that does not impact concurrent operations in any way. Brown *et al.* proposes k-ary search trees [20, 21], which are a generalization of lock-free binary search trees by Ellen *et al.* [31]. Range scans undergo a validation phase for ensuring linearizability and are restarted when a concurrent update is detected. In Jiffy, a range scan may *help* completing some concurrent update operations, and is never restarted. CTrie [49] is a lock-free concurrent hash trie based on CAS. Atomic snapshots are provided through a lazy copy-on-write operation, which slows down concurrent update operations. In CTrie no partial snapshots can be obtained. Minuet [55] is a distributed, in-memory B-tree index with linearizable snapshots. To create snapshots, Minuet also relies on a relatively expensive copy-on-write method, but allows snapshots to be shared across multiple range scans.

Sagonas *et al.* [52, 53, 60] proposed a number of *contention-adapting (CA)* tree-based data structures with linearizable range scans. The data structures feature a lock-based [52, 53] or a lock-free binary search tree [60] with variable-sized *containers* as leaves, implemented as AVL trees, skip lists or immutable data structures that hold multiple key-value entries (similar to revisions in Jiffy). The size of the container is adjusted to the observed contention level (see also Section 3.7). Linearizable range scans are achieved through locking, optimistic scan and validation, or replacing the leaf data structures using CAS with special objects used by concurrent threads to help with completing the range scan (and to block update operations in the meantime). From all of the data structures we discussed so far, only the lock-based variants of the CA trees support batch update operations.

We are aware of several other works on data structures that dynamically adapt to changing contention levels, e.g., [12, 24]. Unlike CA trees, none of the proposed algorithms support linearizable range scans or batch updates.

Some researchers have investigated general techniques for adding linearizable range scans (but not batch updates) to existing concurrent data structures, e.g. [13, 22, 44, 45, 48].

The concurrency control mechanism implemented in Jiffy shares some similarities with the multiversioned transactional engine in [42], which relies on CAS and structures akin to our batch descriptors to make updates atomic. However, unlike this engine, Jiffy is lock-free and no update operation, including batch updates, ever aborts. Crucially, instead of using a shared atomic counter to generate version numbers, Jiffy relies on a system-provided timestamps (CPU's Time Stamp Counter register [39] on x86_64), which greatly reduces contention between concurrent threads on modern 40+ core CPUs. TSC has been used for a similar purpose also in the context of transactional memory [34, 40, 51], a concurrent stack implementation [30], and a serializable (but not linearizable) database engine [43].

3 Design of Jiffy

3.1 The architecture overview

Jiffy is a multiversioned [17] skip list [50], where each *node* (an object on the lowest-level linked list of the skip list) manages a continuous range of keys (see Figure 1). More precisely, each node stores (1) a *node key*, i.e., a key that represents the lower end of the managed key range (the exclusive upper end is defined by the node key of the successor node)[1], and (2) a reference to the head of a *revision list*. The revision list consists of *revisions*, immutable objects, each storing 1-65k (25-300 on average, see Section 4) key-value entries that fit in the node's range (we discuss the layout of data in a revision in Section 3.6). Each revision is tagged with a version number, which is used for all key-value entries stored in the revision. Update operations, such as put, remove or batch update use a copy-on-write technique and the compare-and-swap (CAS)[2] operation to add a new revision as the head of the revision list. The new revision is therefore a modified copy of the previous head of the revision list, such that they differ only on keys modified by the update operation (e.g., for put(k, v), the new revision features a new entry for key k or v replaces the old value for k). We simply say that a new revision that reflects the update has been added to the node. Before each update operation completes, the internal garbage collector is invoked to check if the revision list can be cut short in case certain revisions will not be needed any more.

In Jiffy, *structure modifications*, i.e., changes to the index, are more involved compared to a typical lock-free skip list, such as [27], where nodes are added or removed upon inserting new keys or removing the existing ones. In our approach, the index grows by splitting a node into two and shrinks by merging two nodes into one (see details in Section 3.5). The index starts with a single base node (with key $\perp$) and an empty revision on its revision list (only the base node can have an empty revision). During a split of a node with key n (referred to as node n), a new node n' ($n' > n$), is added directly after node n (or node $\perp$ if the base node undergoes a split). Node n' inherits the upper half of the key range originally assigned to node n (the key of node n does not change). Conversely, during a merge operation of node n, it is merged with the node directly preceding it in the index (i.e., with a node with a strictly lower key; the base node cannot undergo a merge operation and is never removed). The *index nodes* (i.e., the nodes on all but the lowest-level linked lists, which facilitate fast traversals of the data structure) are inserted to the higher-level linked lists probabilistically (in our implementation, the probability of inserting index nodes up to a certain level is the same as in [27]). Operations on higher-level linked lists also rely on CAS and are lock-free.

A node split or a merge can occur only upon some update operation, i.e., put, remove or batch update (see details in Sections 3.2-3.3). When an update operation of some key k is performed and the appropriate node is found (i.e., node n, where $k \geq n$ and there does not exist a node n' where $k \geq n'$), an *autoscaling policy* decides how the update is to be performed (see details in Section 3.7). In majority of cases (99.99% of updates in our tests), a *regular* update is performed. A regular update involves copying the head of the revision list at node n and applying the update on the copied revision (Figure 1b), adding it to the revision list using CAS (Figure 1c), and garbage collecting obsolete revisions, i.e., revisions that will never be read again, including in any snapshot (Figure 1d). Otherwise, a node split or a merge is performed. In the former case, the update to k is reflected in one of the two new *split* revisions (*left split revision* inserted as the head of the revision list on node n and *right split revision* as the head of the revision list on the new node; each node maintains half of the range of the original node). In case of a merge, the new *merge revision* (on the node directly preceding node n in the index) includes the update to k and the entries for all other keys previously stored within the two nodes (and thus has a *left* and a *right* successor). Through split and merge revisions, revision lists are not just simple linked lists, but they branch and join.

Jiffy is a lock-free data structure, which means that it guarantees system-wide progress. To facilitate lock-freedom operations occasionally *help* completing some concurrent update operations. This happens, e.g., when a thread is preempted in the midst of performing an update u on some node n and some other thread either wants to perform an update on node n or read the revision created by the not yet completed operation u.

Since performing an update operation involves a number of steps (especially in case of batch updates or updates that result in node splits or merges), we need to ensure orderly execution of all update operations. Hence we define the following rules:

[1] Unlike in a classic skip list, the first node, called the *base* node, is not just a sentinel but also manages a range of entries (its key is $\perp$, and thus in Figure 1 its key range is $(-\infty, c)$).

[2] CAS$(v, oldV, newV)$ atomically replaces v with $newV$ only if $v = oldV$, and returns a boolean value that indicates if the operation was successful.

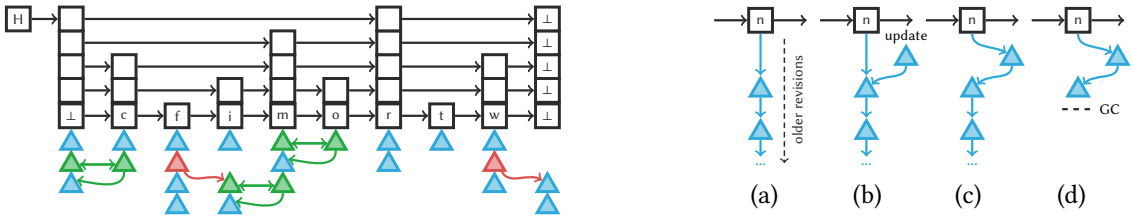

Figure 1. Left: the multiversioned architecture of Jiffy. Each node of the lowest-level list of the skip list manages a range of keys, e.g., $(-\infty, c)$, $[c, f)$, $[f, i)$, etc. Key-value entries are kept in immutable revisions (triangles), each in a concrete version, with newest at the top. The skip list grows and shrinks by splitting or merging nodes and through split (green) and merge (red) revisions. Right: regular update operation: (a) initial state, (b) create a new revision, (c) add the new revision to the node (CAS), (d) garbage collect obsolete revisions.

1. any operation (so also a lookup or a range scan) that during its execution encounters a node split or a merge, always helps completing the operation that invoked the split or merge,
2. an update operation can add a new revision r to the revision list at some node n only if there is no pending operation at node n (before adding r to node n, the update operation helps completing all the pending operations at node n),
3. a batch update (comprising of a set of put and remove operations) starts by updating the highest key in the batch and always continues towards lower keys.

Rule (1) means that our index returns to a stable state (i.e., without ongoing structure changes) as soon as possible, so that subsequent operations (including lookups and range scans) can be performed efficiently. Rules (2) and (3) enforce a consistent order of performing updates (also across batch updates), thus allowing Jiffy to guarantee linearizability. For concurrent operations that update multiple nodes (batch updates and updates that result in node splits or merges) and which access overlapping sets of nodes, rules (2) and (3) give priority to operations that happen on nodes with lower keys. This way we prevent live-locks, which could otherwise happen, when, e.g., there were two threads operating on neighboring nodes, with one constantly attempting to perform a node split and the other a node merge.

The lock-free nature of Jiffy inevitably means that under some highly unfavorable (unlikely) workloads, helping concurrent operations will have a convoying effect which results in all threads attempting to complete the same updates/splits/merges thus wasting resources. This, however, is unavoidable if we are to guarantee system-wide progress.

We include all pseudocodes in the Supplemental Material.

3.2 The put and remove operations

We already briefly stated that Jiffy is a multiversioned data structure. To provide linearizable behavior [37] (intuitively, all operations appear as if they were executed sequentially on a single CPU), threads in a multiversion system typically synchronize on a shared (atomic) counter, which is used to generate version numbers (see, e.g., [16]). This, however, introduces a point of contention that quickly becomes a bottleneck.[3] In Jiffy we avoid such a bottleneck by relying on *system-provided timestamps* (or *timestamps* in short) to generate version numbers. In practice, such timestamps can be acquired extremely efficiently using a high-resolution clock supported by CPU. For example, on the x86_64 architecture, timestamps can be obtained by reading the Time Stamp Counter (TSC), a 64-bit register, which functions as a single CPU-cycle-level resolution wall-clock for the entire multi-CPU machine [25, 39, 51]. TSC is reset to 0 upon machine restart and then advances with constant rate.[4] Reading the TSC register (e.g., using the `RDTSCP` instruction) is an extremely fast operation as it does not involve a system call (in our tests, `RDTSCP` takes about 10 ns to complete).

Since Jiffy is implemented in Java, we do not access TSC directly. Instead, we use the `System.nanoTime()` method [29], which on the popular Java Virtual Machines (JVMs) for the x86_64 platforms, e.g., [9, 10], internally relies on TSC.[5] By specification, `System.nanoTime()` is a thread-safe operation that for all invocations of this method in an instance of JVM returns a monotonically increasing 8B integer.[6]

We use system-provided timestamps (denoted currentTS()) in the following way. Each update operation (put, remove or batch update) and each revision created by such an operation (and added using CAS to some revision list) is associated with two version numbers: initially a temporary one, called an *optimistic version number* and, eventually, a *final version number*, which is set using CAS, and which never changes

[3] Reading the atomic counter is also necessary to create snapshots of the dataset. Our preliminary implementation that used an atomic counter to generate version numbers did not scale past 4-8 threads.

[4] TSC registers across CPU sockets must be synchronized using a synchronous `RESET` signal, which is commonly the case on modern hardware [8, 25].

[5] On other platforms, `System.nanoTime()` is also well optimized. E.g., `System.nanoTime()` from OpenJDK [9] on the new ARM-based Apple Silicon is as fast as on x86_64.

[6] Upon start we record s = `System.nanoTime()` and for each subsequent invocation of `System.nanoTime()` we return `System.nanoTime()` $- s$, so that we can assume that TSC always returns a positive value.

again (see Table 2 for the summary of CAS-related costs of all update operations in Jiffy). An optimistic version number is negative, which signals a concurrent thread that encounters a revision with such a version number about a pending update operation (which the thread might now have to complete). Moreover, there is a special relationship between the optimistic and the final version numbers, which allows us to better handle lookups and range scans performed on snapshots (see Section 3.4).

More precisely, an update commences with an optimistic version number $v = -(t + 1)$, where $t = \text{currentTS}()$. The name *optimistic version number* comes from the fact that $|v|$ corresponds to the lowest possible final version number with which the update operation can complete. Hence we define an invariant $v' \geq |v|$, where v' is the final version number assigned to the revision. For correctness, revisions in each revision list (i.e., revisions linked to the same node) must have unique version numbers. Since system-provided timestamps are guaranteed to be monotonically but not strictly monotonically increasing, we add 1 to t and before we assign the final version number to the revision, we ensure that the current system-provided timestamp is greater or equal v' (we do so through an active wait, albeit in practice it is never used due to TSC's cycle-level resolution). However, any two concurrent updates that modify independent keys (add revisions to different nodes) can be assigned the same final version number, thus enabling unrestricted scalability for such operations.

Note that when we want to remove a key which is not present in the index, no new revision needs to be created. Before each update, the autoscaler is queried to determine whether a node split or merge should be performed. If so, the update is reflected in one of the split revisions or in the merge revision (see details in Section 3.5).

3.3 The batch update operation

A batch update comprises of a number of put and remove operations that are to be performed atomically. The batchUpdate operation in Jiffy relies on the same logic as put and remove, except for a few differences:

1. All put and remove operations to be executed by a batch update are stored in a *batch descriptor*, an additional data structure created for each batch update and referenced by each revision created by the batch update. A batch descriptor also stores a version number (initially an optimistic version number and, eventually, a final version number which is set by CAS). Reading the version number for a revision created by a batch update happens indirectly through the batch descriptor.[7] Therefore the version number is shared by all revisions created by the same batch update.
2. Each revision created by a batch update and added to some node includes all changes from the batch update that pertain to the node's range. We do so to minimize the number of revisions a batch update creates.
3. Execution of a batch update can result in both node splits and node merges, as determined by the autoscaler. E.g., when a batch update adds multiple key-value entries to some node, as a result the node might undergo a split operation.
4. In order to complete a batch update, a thread must add all necessary revisions to appropriate nodes (in descending order of keys) and only then try to assign the final version number to the batch descriptor (using CAS). The same steps apply when a thread helps completing a concurrent batch update.
5. Suppose that a batch update includes the remove(k) operation, and the newest revision r in the appropriate node does not hold a value for key k. Unlike in case of a simple remove(k) operation, where we can return early without modifying the revision list, we need to clone r and add it to the node.

The order of updates performed by a batch update naturally follows from our design assumption for the node merge operation to proceed towards lower keys. Assume that a batch update proceeds in the opposite order (from lower to higher keys). Then it is possible that a batch update adds a revision to some node n, and then proceeds to node n' that directly follows n in the index and decides to perform a node merge operation on n'. Consequently, an additional revision would have to be created on n, which is suboptimal.

Because a batch update proceeds from the higher keys towards lower keys, multiple index traversals might be needed to complete the operation. However, the cost of multiple traversals is negligible compared to the cost of creating and adding revisions to the appropriate nodes.

To explain why cloning a revision is necessary in scenario (5) consider otherwise. Let us assume a concurrent batch update adds a new revision with an update of key k at the same node and finishes with a lower final version number than the batch update from (5). Then a lookup on a snapshot that includes both batch updates would incorrectly return a value for k instead of $\perp$.

The cost of a batch update in terms of the number of CAS operations needed to be performed ranges from 2 (one CAS for adding a single revision that reflects all put/remove operations from the batch update and one CAS for setting the final version number to the batch descriptor) to $n + 1$ (when each put/remove from the batch update requires creating a separate revision), see also Table 2.

3.4 Lookup operations and range scans

On the level of individual nodes, Jiffy uses the same mechanism to facilitate all read operations:

[7] For performance, once the final version v is set to a batch descriptor, v is copied to all appropriate revisions.

Table 2. Cost of performing Jiffy's update operations (node splits/merges may require additional CAS operations to add/remove index nodes to/from the higher-level, sparse linked lists, as in a typical skip-list).

Jiffy operation	CAS operations
put(k,v)*	2
remove(k), k present*	2
remove(k), k not found*	-
batch update, n keys*	min 2, max n+1
one node split/merge during an update	+2

* without node splits or merges

- lookup (get) operations that return the most recent version of the queried key or that operate on a snapshot,
- ascending/descending range scans that are either weakly consistent as in [27] or operate on a snapshot to guarantee linearizability (we focus on the latter ones in the rest of the paper). Range scans are available through Java's standard `Iterator` interface, which means that they return entries one-by-one, not in bulk.

All read operations use the version numbers stored in revisions to retrieve the correct revision (as explained below) and, from it, the value for the searched key. Identifying revisions that belong to a concrete snapshot is possible by associating each snapshot with a *snapshot version* s, which is a system-provided timestamp obtained upon acquiring the snapshot (see also below). A snapshot with snapshot version s corresponds to the state of the dataset at time s.

Let us assume we have traversed the skip list (as in the standard skip list [27, 50]), found the appropriate node and now we evaluate the revisions in its revision list. The most recent value for some key k, as returned by regular get(k), can be found in the most recently completed revision, i.e., the revision with the greatest positive version number. On the other hand, for reads performed on a snapshot with snapshot version s (get(k, s) or issued by a range scan), when evaluating revision r with version number v, we do the following:

- if $|v| > s$, skip reading r,
- if $v > 0 \wedge v \leq s$, and the revision list contains no revision with version number v', s.t. $v < v' \leq s$, then retrieve r, or
- if $v < 0 \wedge -v \leq s$, help completing the pending update operation that created r, resolve the final version number for r, and act accordingly.

Note that lookups or range scans never restart. Read operations performed on a snapshot occasionally help completing concurrent updates and then carry on.

Recall that the revision list can branch and join through split and merge revisions, respectively. Hence, when traversing the revision list to find an entry for key k in a certain version, when encountering a merge revision, we use k to decide whether to proceed to the left or to the right successor of the merge revision. However, a range scan typically intends to retrieve all key-value entries from the appropriate revisions. Hence, if a range scan encounters a merge revision when evaluating a revision list at some node, it retrieves a *bulk revision* that is constructed by recursively traversing all successors of all the encountered merge revisions. In practice, bulk revisions are created extremely rarely. In our tests (see Section 4), revision lists contain at most 3-4 revisions at a time, and usually only 2. Moreover, node merges are rare, so there are few merge revisions that would necessitate in creating bulk revisions.

Upon acquiring a snapshot, it is registered in a special lock-free linked list, so that the inner garbage collector does not dispose off revisions that are still relevant. More precisely, each entry of the list stores a snapshot version s of some snapshot. Jiffy's inner garbage collector periodically scans the list to obtain the lowest such s, so it knows which entries can be safely disposed of. Removing unnecessary revisions happens upon every update operation by cutting the revision list short (recall that revisions appear in the revision list in descending order of their version numbers). A thread can easily refresh the snapshot by querying obtaining a timestamp and writing the new value in the snapshot's entry on the list (no CAS is required, as 8B writes are atomic on x86_64). Note that if a thread wants to use several snapshots at the same time, it suffices that the value s stored in the thread's entry in the snapshot list represents the smallest snapshot version of all thread's snapshots. Hence, the size of the list never exceeds the number of concurrent threads.

3.5 Structure modifications

For simplicity we abstract away from the fact that in Jiffy all structure modifications are streamlined with updates.

Consider the example in Figure 2, which shows how node k that manages a range of keys $[k, r)$ is split in half so a new node o (whose range is $[o, r)$) is inserted between node k and node r. To this end we first create two special revisions, called *left* (*lsr*) and *right split revisions* (*rsr*), which reference each other. Each split revision contains half of the entries from the revision that was the head of the revision list at node k in the beginning. We use CAS to add *lsr* to the revision list at node k (Figure 2b). Next we create a *temporary split node* with key o, whose next pointer is set to r (Figure 2c). We use CAS to swing the next pointer from node k to the temporary split node and thus add it into the index (Figure 2d). Thus, e.g., concurrent lookups searching for keys in range $[o, r)$ will be able to find it and help completing the pending split operation (information necessary to complete the split operation is accessible through split revisions and the temporary split node). Next, we create node o with *rsr* as the sole revision on its revision list. The next pointer of node o is set to node r (Figure 2e). Finally, we use CAS to swing the next pointer

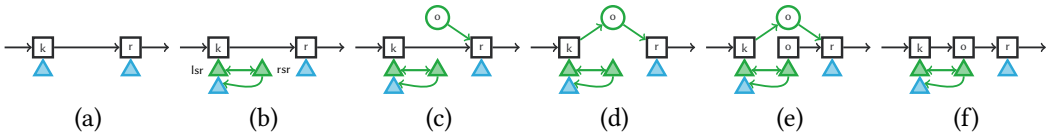

Figure 2. Split operation of node k: (a) initial state, (b) create split revisions (*lsr*, *rsr*), add left split revision *lsr* to node k (CAS), (c) create temporary split node o, (d) add temporary split node o to the index (CAS), (e) create node o with right split revision *rsr*, (f) add node o to the index (CAS) and GC temporary split node o.

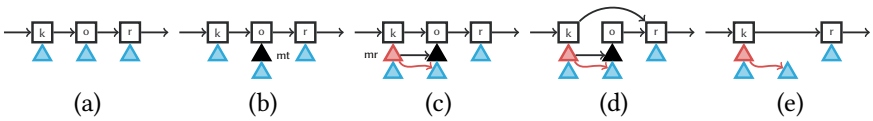

Figure 3. Merge operation of node o: (a) initial state, (b) add merge terminator *mt* to node o (CAS), (c) add merge revision *mr* to node k (CAS), (d) unlink node o from the index (CAS), (e) GC node o and merge terminator *mt*.

of node k from the temporary split node to node o, garbage collect the temporary split node and write the final version number, first to *lsr* and then to *rsr* (Figure 2f).

We cannot simply insert node o in-between nodes k and r using a single CAS operation, as in [35], because the split operation involves adding a revision to node k *and* creating node o. Without a temporary split node an ABA problem is possible. Imagine two threads, T_1 and T_2. T_1 acquires the reference to node r, adds *lsr* to the revision list at node k, and is preempted. Then, T_2 that tries to add a revision at node k, observes a pending split operation and adds node o with *rsr*. Suppose that subsequently node o is merged back to node k, so the next pointer at node k again points to node r. When T_1 resumes its execution, it incorrectly adds node o in-between nodes k and r, which may corrupt lookup and range scan operations. In our scheme the ABA problem on the temporary split node is still possible, but we can recover from it without corrupting concurrent operations. If T_1 observes that some other thread already set the final version number in *lsr* (which is the last operation of a node split), it means that node o must have already been created and merged into node k. In such case the temporary split node can be safely removed.

Now let us consider the node merge operation. In the example in Figure 3b a special (empty) revision called merge terminator (*mt*) is added to the revision list at node o, thus initiating the merge operation. No other revision can now be added to the revision list at node o, hence also a split operation cannot be invoked on node o. Then we find the node directly preceding node o and, if necessary, complete all pending operations at this node (in some cases we need to perform the search for the preceding node again). Once we find node k, we create a merge revision (*mr*) that encompasses the entries from *mt*'s successor revision in the revision list as well as the head of the revision list at node k (Figure 3c). Note that *mr* joins the revision lists at node k and at node o (excluding *mt*), and so has two successors: left (default, same as in a revision added by a regular update) and right. Next we use CAS to swing the next pointer at node k to node r, thus unlinking node o from the index (Figure 3d). Finally, we mark node o as *terminated* (so it can be garbage collected together with the merge terminator) and write the final version number to *mr* (Figure 3e).

In our implementation, structure changes to the index are driven by updates. E.g., a put operation may cause a node split. In such case, one of the split revisions reflects the put operation that caused the node split, so no revisions are created unnecessarily. A remove may cause a node merge. In total, each node split/merge adds 2 CAS operations to the CAS-related cost of performing an update (see Table 2).

3.6 Revision layout

A revision holds two arrays, keys and values, both sorted according to the keys, so that we can perform lookup operations in a cache-friendly manner. Transforming one revision into a new one, as required by update operations, involves copying the arrays and updating/removing the appropriate keys/values. Since all keys and values are kept in a contiguous range of memory, such copy operations are fast.

To speed-up key lookups within a revision, each revision maintains a lightweight hash index that, for each hash value, can store indices to at most two keys from the keys array. Upon lookup of key k, we calculate a 2B hash of k and consult the hash index. For each returned key, we check if it matches k and if so, return the value for k. If neither returned key matches k, we fall back to a binary search on the keys array (k may be in the array but not in the hash index). We return $\perp$ early, when the hash index has no entry for key k or the key referenced by the only entry for key k does not match k. To speed up populating the hash index in new revisions, we cache 2B hashes of keys in each revision. In total, our hash index requires additional 6B for each key-value entry, which is relatively small compared to the typical sizes of key-value entries (see [14, 23] and Section 4.3.2). The parameters of the hash index have been empirically established to be optimal.

3.7 Autoscaling policy

Our preliminary tests showed that, depending on the workload, the revisions should contain between 25 and 300 entries, with larger revisions benefiting read-dominated workloads.[8] Hence we introduce the following *autoscaling policy* to dynamically adjust the sizes of (new) revisions.

Each revision maintains two exponential moving averages pReads and pUpdates that roughly correspond to the amount of time spent by threads performing reads and updates in the revisions in any given node. The times are calculated through differences in system-provided timestamps upon each update and every 100th read (to avoid contention in writing to pReads). For a batch update that created n revisions, we adjust pUpdates on each revision by $1/n$-th of the measured time. The decision to split a node or merge two nodes into one comes from first calculating the writers ratio $\mathsf{wr} = \mathsf{pUpdates}/(\mathsf{pUpdates} + \mathsf{pReads})$. Then, we calculate $\mathsf{optSize} = \max(50, -200\mathsf{wr}+150)$. If the size of the node after a regular update would fall below or exceed the optSize by a factor of 2, we perform the merge or split, respectively. Thus, we effectively limit the size of the node to the range of approximately $[25, 300]$. We use the factor of 2 to avoid repeated splits-and-merges.

Our auto scaling policy keeps balance between readers and writers. On the other hand, the contention-based solutions [52, 53, 60] cater only to writers. In turn, contention-based policies may lead to an uncontrolled growth of the revision sizes in the absence of contention.

3.8 Correctness

Now we argue that Jiffy ensures linearizability [37]. For simplicity, we abstract from node splits and merges. It is easy to see that put, remove (of a key present in the index), and batch update operations (on the same keys) are serialized because: (1) no revision can be added to the revision list if there are pending operations at this node, (2) when encountering a pending operation at some node, a thread helps completing the operation before proceeding with its own update. The final version numbers of revisions in the revision list of each node monotonically decrease when iterating from the head of the revision list (recall that an optimistic version number equals $-v$, where $v = \mathsf{currentTS}() + 1$, so it represents a moment in the future, and the final version number v', which is also a system-provided timestamp, is such that $v' \geq v$). All batch update operations update keys in the descending order of keys. Moreover, for concurrent operations that update multiple nodes and which access overlapping sets of nodes, operations that happen on nodes with lower keys have priority. Thus, no two batch updates with intersecting key sets update revision lists of two nodes in different orders.

The entries created by every update operation can be read by other threads once the final version number is established. The final (positive) version number is written to the revision or to the batch descriptor, using an atomic operation (CAS). The assignment of the final version number is the linearization point for updates that created any revision. Entries created by the same batch update appear as added atomically because they share the version number (through the same batch descriptor). For a remove operation of a key not present in the index, the linearization point occurs when the appropriate revision is accessed and the key is not found.

For a lookup for some key k ($\mathsf{get}(k)$) we observe as follows:

1. Entries (within revisions) for any key k are arranged in the revision list of the node responsible for a key range that includes k, according to their (final) version numbers in descending order (as argued above), and any lookup always evaluates the entries in that order.
2. For each key k, at any moment there can be only a single pending update that modifies k (a revision without the final version number set), which precedes in the revision list all other revisions that might include k.

According to the linearizable semantics, $\mathsf{get}(k)$ must return the newest value written for key k and it may or may not observe the effects of the concurrent operations (operations that have not completed before $\mathsf{get}(k)$ started). The inclusion or exclusion of a concurrent update depends on whether its linearization point lies before or after the one for $\mathsf{get}(k)$. Hence, $\mathsf{get}(k)$ can safely skip reading an entry in a revision whose final version number is not yet determined, and thus return the value from the entry (for key k) from the first revision whose final version number is positive.

Consider now a $\mathsf{get}(k, s)$ operation, where s is a snapshot version. Then the linearization point of the snapshot acquisition or update determines the value returned by $\mathsf{get}(k, s)$. Value s is obtained from TSC upon registering or updating the snapshot. Entries written by updates that finished prior to the acquisition of s have the final version number $v \leq s$ (recall the wait of updates until TSC indicates $t \geq v$). On the other hand, entries written by operations executed concurrently with the snapshot creation/update may (but not necessarily must) have final version numbers $v' > s$. We choose the linearization point for the snapshot creation/update so that it precedes all such concurrent operations.

The $\mathsf{get}(k, s)$ operation chooses the entry for key k from a revision with the greatest final version number $v \leq s$. Recall observation (1). For a revision r with version number v, such that $|v| > s$, we can skip reading r, because if $v < 0$, due to our invariant (see Section 3.2), the final version number for r will be at least $|v|$, so also greater than s. If $v < 0$ and $-v \leq s$, $\mathsf{get}(k, s)$ helps completing the update, i.e., $\mathsf{get}(k, s)$ will be

[8] Jiffy is a generic Java data structure, which means that the arrays in revisions store references to key/value objects, and not the keys/values themselves. Hence, the size of a revision does not depend on the types of keys/values, as could be the case if Jiffy were implemented in, e.g., C++.

able to determine the final version number for this revision. For the first revision with a final (positive) version number $v \leq s$, get(k, s) extracts the value for key k and returns it.

4 Evaluation

4.1 Implementation

Jiffy [5] is implemented in pure Java as a generic concurrent data structure, so it can store key-values of arbitrary types. Jiffy extends Java's `NavigableMap` interface [28], which means that update operations return the overwritten/removed values. This way not only we provide rich interface to the programmer but this additional information also allows us to better check correctness of our implementation (see below).

4.2 Correctness tests

The test harness that we developed to evaluate correctness of Jiffy (and possibly other linearizable concurrent data structures) uses the index operations' return values and timestamps periodically obtained from TSC to construct a graph of dependencies between the operations invoked during a concurrent execution. The keys and values used in the test are specially constructed, so the observed value (returned by an operation) can be traced to the thread that created it and the (logical) moment in the thread's execution when it happened. The edges in the graph correspond to the write-read/write-write dependencies between operations, program and timestamp order, batch update and snapshot-based constraints, etc. We iteratively refine the graph by inferring new relationships between events. If the graph contains a cycle, then the tested implementation definitely is not linearizable.

We tested Jiffy using various mixes of all operations (including batch updates, range scans and lookups on snapshots) with 8-48 concurrent threads and a small keyset (20-200 unique keys) to induce high contention levels. Our tool helped us eliminate several extremely subtle bugs from our implementation. We leave the detailed description of our test harness for future.

4.3 Performance evaluation

Providing rich semantics *and* scalable performance on modern 40+ core CPUs is notoriously difficult. Therefore the goal of our performance evaluation is assessing the multithreaded performance of Jiffy (and its competitors) under varied levels of contention.

4.3.1 Tested indices

We experimentally compared Jiffy with the following state-of-the-art, linearizable ordered indices: SnapTree [19], k-ary tree [20, 21], CA-imm (lock-based contention-adapting tree with immutable containers) [52], CA-AVL and CA-SL (lock-based CA trees with mutable containers based on AVL trees or skip lists) [53] and LFCA tree (lock-free CA tree with immutable containers) [60] (see also Section 2). All indices feature linearizable range scans but only CA-AVL and CA-SL also support linearizable batch updates. For reference, we included `ConcurrentSkipListMap` (Java CSLM) [26], which lacks linearizable range scans and batch updates. In some tests we also included KiWi [16], whose available codebase [6] supports only 4B integer keys.

4.3.2 Benchmark and test environment

We opted for a custom microbenchmark, as there are no off-the-shelf benchmarks that provide extensive support for batch updates. Each microbenchmark thread issues only one type of operations, i.e., either *updates* (puts/removes/batch updates), *lookups* (gets) or *range scans*, so that certain operations, e.g., long-running scans do not stifle the execution of operations of other types. We vary the percentage of threads that perform each kind of operations to uncover the characteristics of all tested indices.

The dataset has the average size of 10M entries (20M unique keys). Jiffy is multiversioned, so it typically maintains more entries at any given moment. The key/value sizes are set to 16/100B and 4/4B, typical for such tests [14, 23]. We examine the systems when keys are randomly chosen with a uniform and a Zipfian distribution ($s = 0.99$, as in [23]). The results are reported in (millions of) *basic* operations per second, i.e., updates or lookups on a single key (a scan over 100 key-value entries counts as 100 get operations).

To assess the performance of updates in Jiffy, we test it in five variants. In the default variant, Jiffy performs all updates as single put or remove operations. Other variants correspond to results obtained when Jiffy executes all update operations in 10-operation batch updates or large, 100-operation batch updates. To demonstrate the performance of batch updates in the extreme cases, they are either *sequential* (update consecutive key-value entries) or *random* (update randomly chosen key-value entries). In a similar way we test CA-AVL and CA-SL, which also support batch updates.

We conducted tests on a server equipped with two Intel Xeon Gold 6252N CPUs, 192 GB of DRAM and running OpenSUSE Tumbleweed with kernel 5.8. Each CPU has 24 cores (48 hyperthreads), is clocked at 2.3 GHz and features 36 MB of L3 cache. We ran our tests on OpenJDK 14.0.2.

4.3.3 Results

We start by discussing the results of tests in which key/value sizes were set to 16/100B and keys were chosen with uniform distribution (see Figure 4). In all tested scenarios, Jiffy exhibits scalable behaviour. Single put/remove operations in Jiffy are slightly more expensive than in SnapTree, CA-imm, LFCA tree or CA-AVL (Figure 4a, top row) because of the multiversioned architecture of our index. In Jiffy each update requires at least two CAS operations: one to add a revision to the revision list at some node and another to set the final version number on the revision itself. In other lock-free indices (that lack support for batch updates) only one CAS

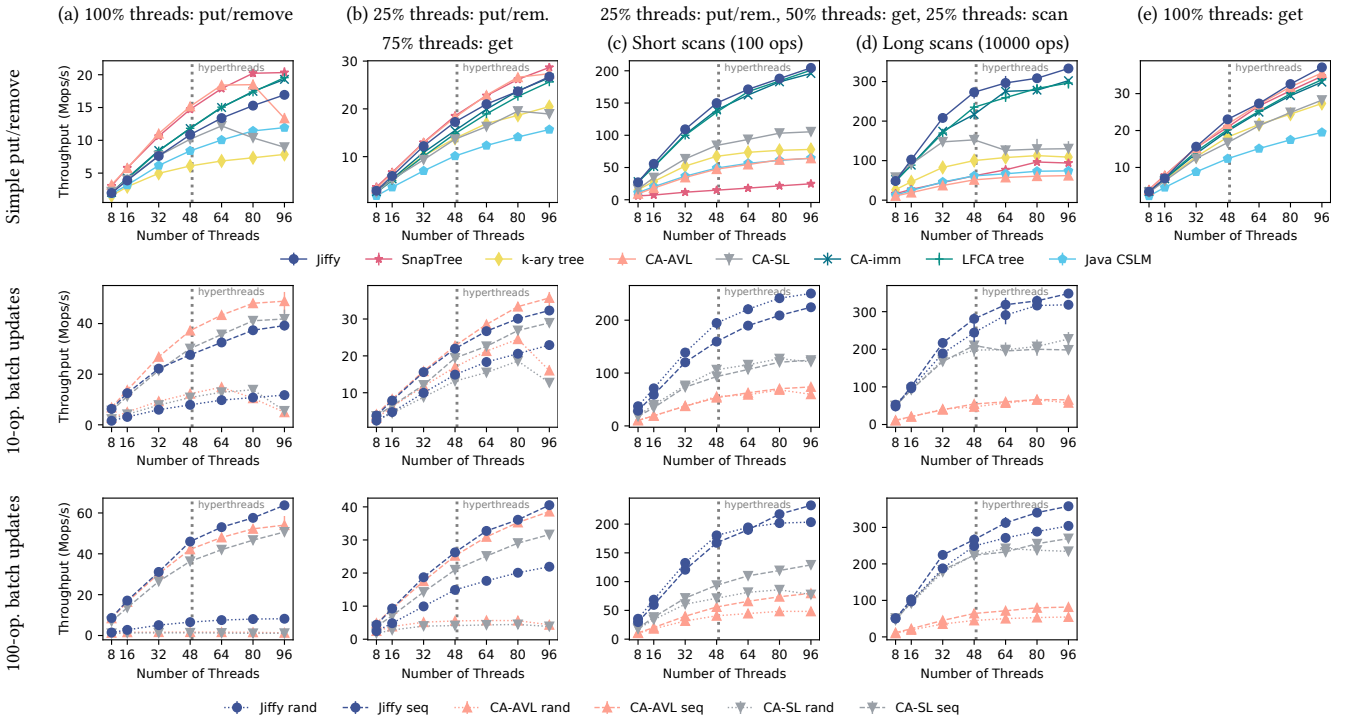

Figure 4. Throughput scalability (16B keys, 100B values, keys chosen with uniform distribution).

is necessary to perform update in-place (e.g., Java CSLM) or to replace an old key-value entry container with a new one (e.g., LFCA tree). In Jiffy there is also small overhead resulting from managing the lightweight hash indices inside revisions. As the hash indices speed-up lookups, the performance differences between Jiffy and the mentioned systems is smaller when lookups are introduced to the workload (see Figure 4b,e). Our autoscaling policy set the revision sizes to ~35 entries in the write-only scenario versus ~130 entries in the update-lookup scenarios. The revision size adjustment time was about 10 second (and 1 second on a 1M entries dataset).

Jiffy executes range scans much more efficiently than its rivals (see Figure 4c-d). LFCA tree and CA-imm are 10% slower than Jiffy's, whereas the only other indices that, similarly to Jiffy, support batch updates, i.e., CA-AVL and CA-SL, at best achieve only half of the total throughput of Jiffy.

Now let us consider the performance of batch updates in Jiffy. When batch updates are small (each includes 10 put/remove operations, Figure 4, middle row), batch updates in Jiffy are slightly slower compared to CA-AVL's and CA-SL's, due to the same reasons, which we discussed earlier when explaining the performance of put/remove operations in Jiffy. Notice that with random batch updates the performance of lock-based CA-AVL and CA-SL starts to diminish towards the higher number of concurrent threads, whereas Jiffy continues to scale thanks to its lock-free architecture. The differences between the lock-based and the lock-free approach start to become apparent when all updates are executed as large batch updates (each includes 100 put/remove operations, bottom row of Figure 4). When batch updates are sequential, the performance of Jiffy is about 15% better than the performance of either CA-AVL or CA-SL (in the write-only scenario). However, with random batch updates, Jiffy's maximal throughput is even 4.9× and 6.1× of the maximal throughput of CA-AVL and CA-SL, respectively.

Notice the surprising way in which small batch updates impact Jiffy's performance in the mixed scenario with small range scans (Figure 4c, middle row). Using random batch updates results in a slightly better overall performance, compared to sequential batch updates, which are on average much cheaper to execute (with n puts/removes in a batch update, the former creates n revisions on average versus $1-2$ for the latter). This phenomenon can be explained by examining the throughput of updates (for additional plots see the Supplemental Material): with small sequential batch updates, Jiffy executes 4× as many updates compared to the same test with random batch updates. In turn, in the former test, Jiffy has to manage many more revisions, which translates into slightly worse performance of lookups and scans.

With 4B key/value sizes and uniform key distribution (see Figure 5), in the scenarios (a) and (b) KiWi (whose implementation is optimized for 4B integer keys and does not support other key/value types) only narrowly beats other indices. Overall, the relative differences between the performance of the tested indices stay largely the same, except for two small

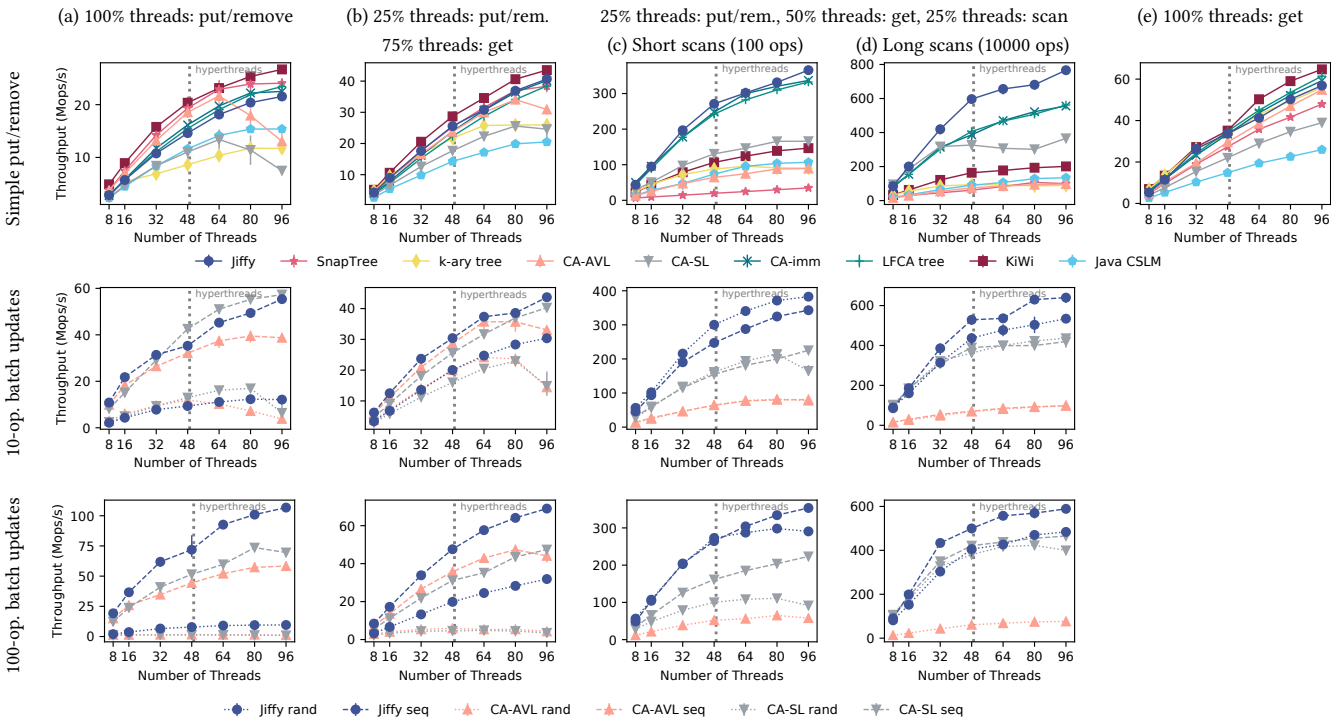

Figure 5. Throughput scalability (4B keys, 4B values, keys chosen with uniform distribution).

differences. Firstly, with smaller key/value sizes, the performance of lock-based CA-AVL and CA-SL starts to diminish with a smaller number of concurrent threads. Secondly, we observe a much more substantial advantage of Jiffy in workloads that feature range scans. In the mixed scenario with long range scans, Jiffy beats the second-best performing indices CA-imm and LFCA tree by 30%. Moreover, with large batch updates, Jiffy's advantage over CA-AVL and CA-SL is even more pronounced (the speedup over rivals increases to 1.5× for sequential batches and from 4.9×/6.1× (for 8 threads) to 5.7×/7.4× (for 96 threads) for random batch updates, for CA-AVL and CA-SL, respectively).

Below we summarize the results of tests conducted with skewed workloads (see the Supplemental Material for details). A skewed workload results in much higher contention, which is further amplified when batch updates are used, each creating many new revisions (containers in CA-AVL and CA-SL, see Figure 7 and Figure 9 in the Supplemental Material). Such workloads were almost equally bad for Jiffy and its lock-based competitors. In the write-only scenario, the observed throughput for Jiffy, CA-AVL and CA-SL was about 1.5-2 Mops/s for small random batch updates and 0.3-0.5 Mops/s for large random batch updates.

4.3.4 Summary

Compared to its rivals, Jiffy achieves remarkable performance and scalability across different workloads. It is so even though update operations in Jiffy are more expensive than in other systems (typically each update operation in Jiffy requires two CAS operations whereas the other tested indices require only one CAS operation per update). This increased cost of update operation cannot be avoided if Jiffy is to provide atomic batch updates. However, the fully-fledged multiversioning concurrency control mechanism allows Jiffy to provide a robust snapshot mechanism that is much more flexible than in its competitors.

The impressive scalability of Jiffy could not be achieved without relying on system-provided timestamps to generate version numbers, as we discussed in Section 3.2. The preliminary version of Jiffy that kept all key-value entries in separate nodes (instead of multi-entry revisions) suffered from the same bottleneck as `ConcurrentSkipListMap` [27] (Java CSLM in our tests), mainly a relatively large index which is time-consuming to traverse. Adding hash indices to revisions slightly increased the cost of updates due to additional arrays that need to be copied during creation of new revisions, but significantly sped up reads and also made Jiffy's performance more consistent across workloads. In none of our tests, Java's GC was a bottleneck.

5 Conclusions

Jiffy is the first lock-free, linearizable ordered key-value index with batch updates and snapshots (source code is available [5]). Despite its rich functionality, Jiffy offers scalable performance and, e.g., with mixed workloads (updates, lookups and range scans) it outperforms the state-of-the-art indices

with less flexible semantics. Crucially, Jiffy's novel lock-free, multiversioned algorithm allows it to execute batch updates more efficiently compared to its lock-based, less scalable rivals, with speedup in throughput reaching 7.4×, making it an extremely versatile concurrent data structure.

Acknowledgments

This work was supported by the Foundation for Polish Science, within the TEAM programme co-financed by the European Union under the European Regional Development Fund, grant No. POIR.04.04.00-00-5C5B/17-00. We thank Intel Corporation and Intel Technology Poland for providing us with computing resources equipped with the Intel® OptaneTM DC persistent memory technology. We also thank reviewers for their thorough review of the manuscript and the constructive suggestions.

References

[1] [n.d.]. Apache Hadoop. https://github.com/apache/hadoop.
[2] [n.d.]. Apache Spark. https://github.com/apache/spark.
[3] [n.d.]. Bitcoin. https://github.com/bitcoin/bitcoin.
[4] [n.d.]. Couchbase. https://www.couchbase.com.
[5] [n.d.]. Jiffy. https://github.com/tkobus/jiffy.
[6] [n.d.]. KiWi. https://github.com/sdimbsn/KiWi.
[7] [n.d.]. LevelDB. https://github.com/google/leveldb.
[8] [n.d.]. Linux Kernel 5.8 source code. https://elixir.bootlin.com/linux/v5.8/source/arch/x86/kernel/cpu/intel.c#L243.
[9] [n.d.]. OpenJDK. https://openjdk.java.net.
[10] [n.d.]. Oracle Java Development Kit. https://www.oracle.com/technetwork/java/.
[11] [n.d.]. RocksDB. https://github.com/facebook/rocksdb.
[12] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. 2012. CBTree: A Practical Concurrent Self-Adjusting Search Tree. In *Proc. of DISC '12*. 1–15.
[13] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proc. of PPoPP '18*. 14–27.
[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. of SIGMETRICS '12*. 53–64.
[15] Hillel Avni, Nir Shavit, and Adi Suissa. 2013. Leaplist: Lessons Learned in Designing Tm-Supported Range Queries. In *Proc. of PODC '13*. 299–308.
[16] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *Proc. of PPoPP '17*. 357–369.
[17] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (Dec. 1983).
[18] Anastasia Braginsky and Erez Petrank. 2012. A Lock-free B+Tree. In *Proc. of SPAA '12*. 58–67.
[19] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *Proc. of PPoPP '10*. 257–268.
[20] Trevor Brown and Hillel Avni. 2012. Range Queries in Non-blocking k-ary Search Trees. In *Proc. of OPODIS '12*. 31–45.
[21] Trevor Brown and Joanna Helga. 2011. Non-Blocking k-Ary Search Trees. In *Proc. of OPODIS '11*. 207–221.
[22] Bapi Chatterjee. 2017. Lock-Free Linearizable 1-Dimensional Range Queries. In *Proc. of ICDCN '17*.
[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of SoCC '10*. 143–154.
[24] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. A Contention-Friendly Binary Search Tree. In *Proc. of Euro-Par '13*.
[25] Martin G. Dixon, Jeremy J. Shrall, and Rajesh S. Parthasarathy. 2011. Controlling Time Stamp Counter (TSC) Offsets For Mulitple Cores And Threads. USPTO patent no. US 20110154090 A1, Jun. 23, 2011.
[26] Java documentation. [n.d.]. Java concurrent collections. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html.
[27] Java documentation. [n.d.]. Java ConcurrentSkipListMap. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ConcurrentSkipListMap.html.
[28] Java documentation. [n.d.]. Java NavigableMap interface. https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html.
[29] Java documentation. [n.d.]. Java `System.nanotime()`. https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime().
[30] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. In *Proc. of POPL '15*. 233–246.
[31] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proc. of PODC '10*. 131–140.
[32] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-Free Linked Lists and Skip Lists. In *Proc. of PODC '04*. 50–59.
[33] K. Fraser. 2004. *Practical Lock-freedom*. Ph.D. Dissertation. University of Cambridge.
[34] Ellis Giles, Kshitij Doshi, and Peter Varman. 2018. Hardware Transactional Persistent Memory. In *Proc. of MEMSYS '18*. 190–205.
[35] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proc. of DISC '01*. 300–314.
[36] M. Herlihy. 1990. A Methodology for Implementing Highly Concurrent Data Structures. In *Proc. of PPoPP '90*. 197–206.
[37] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
[38] Shane V. Howley and Jeremy Jones. 2012. A Non-Blocking Internal Binary Search Tree. In *Proc. of SPAA '12*. 161–171.
[39] Intel Corporation 2008. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation.
[40] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with TimeStone. In *Proc. of ASPLOS '20*. 335–349.
[41] Sarath Lakshman, Sriram Melkote, John Liang, and Ravi Mayuram. 2016. Nitro: A Fast, Scalable In-Memory Storage Engine for NoSQL Global Secondary Index. *Proc. of VLDB Endowment* 9, 13 (Sept. 2016), 1413–1424.
[42] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. of VLDB Endowment* 5, 4 (Dec. 2011), 298–309.
[43] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proc. of SIGMOD '17*. 21–35.
[44] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-Log-Update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proc. of SOSP '15*. 168–183.
[45] Paul E. McKenney and John D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of PDCS '98*. 509–518.
[46] Maged M. Michael and Michael L. Scott. 1995. *Correction of a Memory Management Method for Lock-Free Data Structures*. Technical Report.
[47] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proc. of PPoPP '14*. 317–328.

[48] Erez Petrank and Shahar Timnat. 2013. Lock-Free Data-Structure Iterators. In *Proc. of DISC '13*. 224–238.

[49] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proc. of PPoPP '12*. 151–160.

[50] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676.

[51] Wenjia Ruan, Yujie Liu, and Michael Spear. 2013. Boosting Timestamp-Based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Transactions on Architecture and Code Optimization* 10, 4 (Dec. 2013), 40:1–40:21.

[52] Konstantinos Sagonas and Kjell Winblad. 2015. Contention Adapting Search Trees. In *Proc. of ISPDC '15*. 215–224.

[53] Konstantinos Sagonas and Kjell Winblad. 2018. A contention adapting approach to concurrent ordered sets. *J. Parallel and Distrib. Comput.* 115 (2018), 1–19.

[54] Niloufar Shafiei. 2013. Non-blocking Patricia Tries with Replace Operations. In *Proc. of ICDCS '13*. 216–225.

[55] Benjamin Sowell, Wojciech Golab, and Mehul A. Shah. 2012. Minuet: A Scalable Distributed Multiversion B-Tree. *Proc. of VLDB Endowment* 5, 9 (May 2012), 884–895.

[56] Michael Spiegel and Paul F. Reynolds Jr. 2010. Lock-Free Multiway Search Trees. In *Proc. of ICPP '10*. 604–613.

[57] Håkan Sundell and Philippas Tsigas. 2003. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *Proc. of IPDPS '03*. 84–96.

[58] Håkan Sundell and Philippas Tsigas. 2004. Scalable and Lock-Free Concurrent Dictionaries. In *Proc. of SAC '04*. 1438–1445.

[59] John D. Valois. 1995. Lock-Free Linked Lists Using Compare-and-Swap. In *Proc. of PODC '95*. 214–222.

[60] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. 2018. Lock-Free Contention Adapting Search Trees. In *Proc. of SPAA '18*. 121–132.

A Artifact Appendix

A.1 Abstract

We provide the source code of Jiffy and our test harness for evaluating correctness of linearizable concurrent data structures. We also describe how the experiments discussed in the paper can be rerun and the obtained results compared to the ones presented in Section 4.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Jiffy lock-free ordered key-value index.
- **Compilation:** Java 11+, Gradle 6.5+, Python 3 with `matplotlib`, LaTeX.
- **Data set:** Randomly generated workloads.
- **Hardware:** A single multi-core machine (originally tested on a two-socket Intel machine, each socket consisting of 24 hyperthreaded cores, see Section 4.3.2).
- **Metrics:** Indices' throughput in the function of the number of concurrent threads.
- **Output:** Data files and figures.
- **Experiments:** Scripts and instructions to fully reproduce the paper's results are provided in the artifact README files.
- **How much time is needed to complete experiments (approximately)?:**
  - Correctness tests–a single test with default configuration parameters completes in up to 30 minutes (on a 16+ core machine; a machine with fewer cores will complete the test in a shorter time, because with restricted concurrency, there is less data to be processed).
  - Performance tests–5h to obtain results for Jiffy, as presented in the main paper, 4 days to obtain results for all indices, as required to reproduce all figures in the main paper and the supplemental material.
- **Publicly available?:** Yes [5] (see also Appendix A.3.1).
- **Code licenses (if publicly available)?:** MIT.

A.3 Description

A.3.1 How to access

For Jiffy's source code see the main Jiffy repository [5] (also available as a Zenodo archive: https://doi.org/10.5281/zenodo.5733227). Due to legal issues (the lack of licenses for some systems we tested Jiffy against), currently the repository does not include the performance evaluation framework (which has been made available to the reviewers and which can be provided by the authors on request).

A.3.2 Hardware dependencies

Jiffy is a Java-based library, which means it can be used on any JVM-supported machine. Since the focus of the performance evaluation has been the scalability of Jiffy, a multi-socket, multi-core (40+) server, with adequate (40+ GB) DRAM setup is desired. See our hardware setup in Section 4.3.2.

A.3.3 Software dependencies

Jiffy is written in pure Java (compatible Java 11+) and relies on Gradle 6.5+ scripts for compilation. Performance tests are available through Bash and Python scripts. The generated output files are converted through Python scripts to plots. Finally, we use LaTeX to arrange the plots into figures, as found in the paper.

A.3.4 Data sets

We use randomly generated workloads of predefined characteristics (see Sections 4.2 and 4.3.2).

A.4 Installation

Detailed instructions on using and running the code, and obtaining the plots are included in the artifact's README file. We also discuss how the performance tests can be expedited when one has multiple machines available.

A.5 Experiment workflow

A.5.1 Correctness tests

The test harness that we developed to evaluate correctness of Jiffy is provided as part of the Jiffy library. Instructions on how to run the test (possibly with other configuration parameters) is given in the README file.

Each correctness test comprises of a few seconds long warmup followed by a short run on a randomized concurrent workload that includes a mix of all operations supported by Jiffy (so put/remove/batch updates, lookups, ascending and descending range scans that are performed on snapshots, etc.). The logs of the performed operations are then iteratively processed and checked against a set of conditions that should be respected if the tested data structure is indeed linearizable.

A.5.2 Performance tests

The benchmark scripts allow one to run randomized workloads of predefined characteristics on Jiffy and its competitors. The output files for a given index and test scenario include, for each test round, the measured throughput (as well as numerous other statistics, such as detailed update/lookup/range scan operations count) in the function of the number of concurrent threads. The performance tests can be easily partitioned to run on multiple machines concurrently.

After test runs complete, a single script can be used to generate plots from the raw data files. This task includes using LaTeXto produce figures that can be directly compared to the ones included in the paper.

A.6 Evaluation and expected results

A.6.1 Correctness tests

All assertions included in the test harness should pass. Below we outline some of the checked conditions:

- no two read-only operations on the same key performed on the same snapshot return two different values,
- program order of operation execution is satisfied,
- there exists a total-order of update operations on each key,
- given the inferred time-based constraints, real-time order of operation execution is satisfied,
- the graph of operations (with nodes corresponding to the performed operations and periodically taken timestamps and edges corresponding to the write-read/write-write dependencies between operations and time-based relationships inferred between operations) is acyclic.

Naturally, a single successfully completed test does not prove that the tested index is correct. Prior paper submissions, we have run the correctness tests in a loop for days at a time to discover bugs in our code.

A.6.2 Performance tests

Provided similar hardware (see our hardware setup in Section 4.3.2), users are expected to reproduce the results presented in this paper.

Note that the performance of concurrent data structures is highly sensitive to the hardware used, such as the CPU architecture and instruction set (x86_64 vs ARM and, in consequence, e.g., the used cache coherence protocols), the CPU configuration (single-socket vs multi-socket), the speed and sizes of CPU caches.

Naturally the use of a concrete JVM also can impact the performance of the tested indices. However, our experience shows that with the popular JVMs [9, 10], the observed differences are not significant. Much more pronounced effects may have changes to the JVM configuration, such as adjustments of the heap size (not always a larger heap is better), or the choice of the garbage collector algorithm.

A.7 Experiment customization

A detailed discussion on altering the benchmark parameters can be found in the artifact's README file.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html