KiWi: A Key-Value Map for Scalable Real-Time Analytics

DMITRY BASIN, EDWARD BORTNIKOV, and ANASTASIA BRAGINSKY, Yahoo Research, Oath, Haifa, Israel
GUY GOLAN-GUETA, VMWare Research Group, Tel Aviv, Israel
ESHCAR HILLEL, Yahoo Research, Oath, Haifa, Israel
IDIT KEIDAR, Technion and Yahoo Research, Oath, Haifa, Israel
MOSHE SULAMY, Tel Aviv University, Tel Aviv, Israel

Modern big data processing platforms employ huge in-memory *key-value* (KV) maps. Their applications simultaneously drive high-rate data ingestion *and* large-scale analytics. These two scenarios expect KV-map implementations that scale well with both real-time updates and large atomic scans triggered by range queries.

We present KiWi, the first atomic KV-map to efficiently support simultaneous large scans and real-time access. The key to achieving this is treating scans as first class citizens, and organizing the data structure around them. KiWi provides wait-free scans, whereas its put operations are lightweight and lock-free. It optimizes memory management jointly with data structure access. We implement KiWi and compare it to state-of-the-art solutions. Compared to other KV-maps providing atomic scans, KiWi performs either long scans or concurrent puts an order of magnitude faster. Its scans are twice as fast as *non-atomic* ones implemented via iterators in the Java skiplist.

ACM Reference format:
Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2020. KiWi: A Key-Value Map for Scalable Real-Time Analytics. *ACM Trans. Parallel Comput.* 1, 1, Article 1 (January 2020), 26 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Motivation and goal. The ordered *key-value* (KV) map abstraction has been recognized as a popular programming interface since the dawn of computer science, and remains an essential component of virtually any computing system today. It is not surprising, therefore, that with the advent of multi-core computing, many scalable concurrent implementations have emerged, e.g., [6, 11, 12, 22, 27, 30].

KV-maps have become a centerpiece to web-scale data processing systems such as Google's F1 [33], which powers its AdWords business, and Yahoo's Flurry [4] – the technology behind Mobile Developer Analytics. For example, as of early 2016, Flurry reported systematically collecting data of 830,000 mobile apps [1] running on 1.6 billion user devices [2]. Flurry streams this data into a massive index, and provides a wealth of reports over the collected data. Such *real-time analytics* applications push KV-store scalability requirements to new levels and raise novel use cases. Namely, they require both (1) low latency ingestion of incoming data, and (2) high performance analytics of the resulting dataset.

The stream scenario requires the KV-map to support fast *put* operations, whereas analytics relies on (typically large) *scans* (i.e., range queries). The consistency (atomicity) of scans is essential for correct analytics. The new challenge that arises in this environment is allowing consistent scans to be obtained *while the data is being updated in real-time.*

2020. 1539-9087/2020/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We present KiWi, the first KV-map to efficiently support large atomic scans as required for data analytics alongside real-time updates. Most concurrent KV-maps today do not support atomic scans at all [6, 9, 11, 12, 22, 27, 28, 30]. A handful of recent works support atomic scans in KV-maps, but they either hamper updates when scans are ongoing [14, 32], or fail to ensure progress to scans in the presence of updates [15]. See Section 2 for a discussion of related work.

The emphasis in KiWi's design is on facilitating synchronization between scans and updates. Since scans are typically long, our solution avoids livelock and wasted work by always allowing them to complete (without ever needing to restart). On the other hand, updates are short (since only single-key puts are supported), therefore restarting them in cases of conflicts is practically "good enough" – restarts are rare, and when they do occur, little work is wasted. Formally, KiWi provides *wait-free* gets and scans and *lock-free* puts.

Design principles. To support atomic wait-free scans, KiWi employs multi-versioning [10]. But in contrast to the standard approach [26], where each put creates a new version for the updated key, KiWi only keeps old versions that are needed for ongoing scans, and otherwise overwrites the existing version. Moreover, version numbers are managed by scans rather than updates, and put operations may overwrite data without changing its version number. This unorthodox approach offers significant performance gains given that scans typically retrieve large amounts of data and hence take much longer than updates. It also necessitates a fresh approach to synchronizing updates and scans, which is a staple of KiWi's design.

A second important consideration is efficient memory access and management. Data in KiWi is organized as a collection of *chunks*, which are large blocks of contiguous key ranges. Such data layout is cache-friendly and suitable for non-uniform memory architectures (NUMA), as it allows long scans to proceed with few fetches of new data to cache or to local memory. Chunks regularly undergo maintenance to improve their internal organization and space utilization (via *compaction*), and the distribution of key ranges into chunks (via splits and merges). KiWi's *rebalance* abstraction performs batch processing of such maintenance operations. The synchronization of rebalance operations with ongoing puts and scans is subtle, and much of the KiWi algorithm is dedicated to handling possible races in this context.

Third, to facilitate concurrency control, we separate chunk management from indexing for fast lookup: KiWi employs an *index* separately from the (chunk-based) data layer. The index is updated lazily once rebalancing of the data layer completes.

Finally, KiWi is a balanced data structure, providing logarithmic access latency in the absence of contention. This is achieved via a combination of (1) using a balanced index for fast chunk lookup and (2) partially sorting keys in each chunk to allow for fast in-chunk binary search. The KiWi algorithm is detailed in Section 3 and we prove its correctness in Section 4.

Evaluation results. KiWi's Java implementation is available in github[1]. In Section 5 we benchmark it under multiple representative workloads. In the vast majority of experiments, it significantly outperforms existing concurrent KV-maps that support scans. KiWi's advantages are particularly pronounced in our target scenario with long scans in the presence of concurrent puts, where it not only performs *all* operations faster than the competitors [14, 15], but actually executes either updates or scans an order of magnitude faster than every other solution supporting atomic scans. Notably, KiWi's atomic scans are also two times faster than the *non-atomic* ones offered by the Java Skiplist [6].

[1]https://github.com/sdimbsn/KiWi

2 RELATED WORK

Techniques. KiWi employs a host of techniques for efficient synchronization, many of which have been used in similar contexts before. Multi-versioning [10] is a classical database approach for allowing atomic scans in the presence of updates, and has also been used in the context of transactional memory [26]. In contrast to standard multi-versioning, KiWi does not create a new version for each update, and leaves version numbering to scans rather than updates.

Braginsky and Petrank used lock-free chunks for efficient memory management in the context of non-blocking linked lists [11] and B$^+$trees [12]. However, these data structures do not support atomic scans as KiWi does.

KiWi separates index maintenance from the core data store, based on the observation that index updates are only needed for efficiency and not for correctness, and hence can be done lazily. This observation was previously leveraged, e.g., for a concurrent skip list, where only the underlying linked list is updated as part of the atomic operation and other links are updated lazily [13, 23, 24, 35].

	scans				**performance**	
	atomic	**multiple**	**partial**	**wait-free**	**balanced**	**fast puts**
Ctrie [32]	✓	✓	✗	✗	✓	✗
SnapTree [14]	✓	✓	✓	✗	✓	✗
k-ary tree [15]	✓	✓	✓	✗	✗	✓
snapshot iterator [31]	✓	✗	✗	✓	✓	✓
Java skiplist [6]	✗	✓	✓	✓	✓	✓
KiWi	**✔**	**✔**	**✔**	**✔**	**✔**	**✔**

Table 1. Comparison of concurrent data structures implementing scans. For range queries, support for multiple partial scans is necessary. Fast puts do not hamper updates (e.g., by cloning nodes) when scans are ongoing.

Concurrent maps supporting scans. Table 1 summarizes the properties of state-of-the-art concurrent data structures that support scans, and compares them to KiWi. SnapTree [14] and Ctrie [32] use lazy copy-on-write for cloning the data structure in order to support snapshots. This approach severely hampers put operations when scans are ongoing, as confirmed by our empirical results for SnapTree, which was shown to outperform Ctrie. Moreover, in Ctrie, partial snapshots cannot be obtained.

Brown and Avni [15] introduced range queries for the k-ary search tree [16]. Their scans are atomic and lock-free, and outperform those of Ctrie and SnapTree in most scenarios. However, each conflicting put restarts the scan, degrading performance as scan sizes increase. Additionally, k-ary tree is unbalanced; its performance plunges when keys are inserted in sequential order (a common practical case).

Some techniques offer generic add-ons to support atomic *snapshot iterator* in existing data structures [18, 31]. However, [31] supports only one scan at a time, and [18]'s throughput is lower than k-ary tree's under low contention.

Most concurrent key-value maps do not support atomic scans in any way [6, 11, 12, 22, 27, 28, 30]. Standard iterators implemented on such data structures provide non-atomic scans. Among these, we compare KiWi to the standard Java concurrent skip-list [19].

Distributed KV-maps. Production systems often exploit persistent KV-stores like Apache HBase [3], Google's Bigtable [17], and others [7, 8]. These technologies combine on-disk indexes for persistence with an in-memory KV-map for real-time data acquisition. They often support atomic scans, which

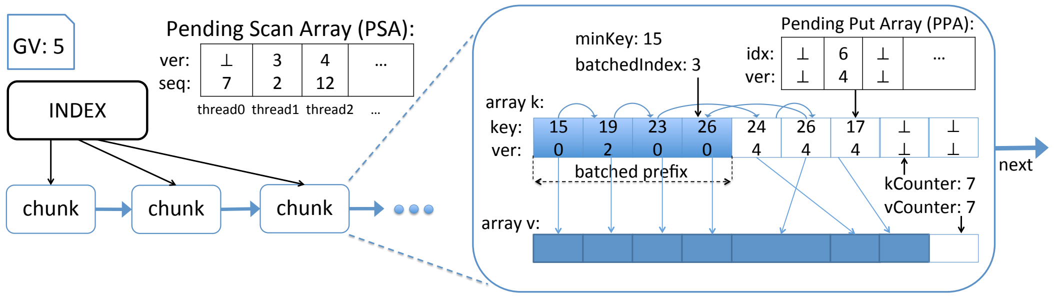


Fig. 1. KiWi data structure layout. In the zoomed in chunk (on the right), a pending put by the second thread is attempting to add k[6] to the linked list with key 17 and version 4.

can be non-blocking as long as they can be served from RAM [20]. However, storage access is a principal consideration in such systems, which makes their design different from that of in-memory stores as discussed herein.

MassTree [29] is a persistent B$^+$-tree designed for high concurrency on SMP machines. It is not directly comparable to KiWi as it does not support atomic snapshots, which is our key emphasis. Sowell et. al. [34] presented Minuet – a distributed in-memory data store with snapshot support. In that context, snapshot creation is relatively expensive, which Minuet mitigates by sharing snapshots across queries.

3 KIWI ALGORITHM

KiWi implements a concurrent ordered key-value map supporting atomic (linearizable) get(key), put(key,value), and scan(fromKey,toKey) operations. Its put operations are lock-free, whereas get and scan are wait-free. A put with a non-existent key creates a new KV-pair, and a put of the ⊥ value removes the pair if the key exists.

The philosophy behind KiWi is to serve client operations quickly, while deferring data structure optimizations to a maintenance procedure that runs infrequently. The maintenance procedure, *rebalance*, balances KiWi's layout so as to ensure fast access, and also eliminates obsolete information.

In Section 3.1 we explain how data is organized in KiWi. Section 3.2 discusses how the different operations are implemented atop this data structure in the common case, when no maintenance is required. Section 3.3 focuses on rebalancing.

3.1 Data organization

Chunk-based data structure. Similarly to a B$^+$tree, the KiWi data structure is organized as a collection of large blocks of contiguous key ranges, called *chunks*. Organizing data in such chunks allows memory allocation/deallocation to occur infrequently. It also makes the design cache-friendly and appropriate for NUMA, where once a chunk is loaded to local memory, access time to additional addresses within the chunk is much shorter. This is particularly important for scans, which KiWi seeks to optimize, since they access contiguous key ranges, often residing in the same chunk.

The KiWi data layout is depicted in Figure 1, with one chunk zoomed in. The chunk data structure is described in Algorithm 1.

KiWi's chunks are under constant renewal, as the rebalance process removes old chunks and replaces them with new ones. It not only splits (over-utilized) and merges (under-utilized) chunks

Algorithm 1 KiWi chunk data structure.

```
immutable minKey                              ▷ minimal key in chunk
array k of ⟨key, ver, valPtr, next⟩           ▷ in-chunk linked list, $k.size > 2$·num_threads
array v of values                             ▷ values stored in the list, $v.size > 2$·num_threads
kCounter, vCounter                            ▷ end of allocated (full) prefixes
batchedIndex                                  ▷ end of batched prefix in k
array ppa[NUM_THREADS] of ⟨ver, idx⟩          ▷ pending put array allowing scans & gets to help puts
next                                          ▷ pointer to next chunk in chunk list
mark                                          ▷ indicates whether next is immutable
rebalance data ⟨status, parent, ro⟩           ▷ rebalancing-related data
```

as in a B$^+$tree, but also improves their internal organization, performs *compaction* by eliminating obsolete data, and may involve any number of chunks.

In order to simplify concurrency control, however, we do not organize chunks in a B$^+$tree, but rather in a sorted linked list. This eliminates the synchronization complexity of multi-level splits and merges. Yet, to allow fast access, we supplement the linked list with an auxiliary *index* that maps keys to chunks; it may be organized in an arbitrary way (e.g., skip-list or search tree). Each chunk is indexed according to the minimal key it holds, which is invariant throughout its lifespan. (The minimal key of the first chunk in the list is $-\infty$.) The index supports a wait-free lookup method that returns the indexed chunk mapped to the highest key that does not exceed a given key. It further supports conditional updates, which are explained in Section 3.3, as they are done only as part of the rebalance procedure. Such updates are lazy, and so the index may be inaccurate. Therefore, the index-based search is supplemented by a traversal of the chunk linked list.

Intra-chunk organization. Each chunk is organized as an array-based linked list, sorted in increasing key order. KiWi chunks hold two arrays – v with written values, and k with the linked list. Both arrays hold more entries than twice the number of threads, to ensure that a chunk does not fill up with only pending operations (in case all threads but one become idle). Each cell in k holds a key, a pointer valPtr to a value in v, and the index of the cell in k that holds the next key in the linked list. It also has a version number, as we explain shortly. When a chunk is created (as a result of rebalancing), some prefix (typically one half) of each array contains data, and the suffix consists of empty entries for future allocation.

The chunk's full prefix is initialized as sorted, that is, the linked-list successor of each cell is the ensuing cell in k. The sorted prefix is called the chunk's *batched prefix*, and it can be searched efficiently using binary search. As keys are added, the order in the remainder of the chunk is not preserved, i.e., the batched prefix usually does not grow. For example, when a key is inserted to the first free cell, it creates a *bypass* in the sorted linked list, where some cell i in the batched prefix points to the new cell, and the new cell points to cell $i + 1$. We note that in case the insertion order is random, inserted cells are most likely to be distributed evenly in between the batched prefix cells, thus creating fairly short bypasses. Given that the prefix and the remainder are of similar sizes, the expected search time remains poly-logarithmic. Nevertheless, in the worst-case, the search time is linear in the size of the remainder of the chunk.

In order to support atomic scans, KiWi employs *multi-versioning*, i.e., sometimes keeps the old version of a key instead of overwriting it. To this end, KiWi maintains a *global version*, GV, and tags each key-value pair with a version, ver. Versions of a key are chained in the linked-list in descending version order, so the most recent version is encountered first. The compaction process that occurs as part of rebalancing eliminates obsolete versions. Unlike traditional multi-versioning, KiWi creates new versions *only* as needed for ongoing scans. This allows us to shift the overhead for version management from updates, which are short and frequent, to scans, which are typically long and therefore much less frequent. Specifically, put operations continue to use the same version (overwriting previous values for written keys, if they exist) as long as no scan operation increases GV.

Coordination data structures. KiWi employs two data structures for coordinating different operations. A global *pending scan array* (PSA) tracks versions used by pending scans for compaction purposes; each entry consists of a version ver and a sequence number seq, as well as the scan's key range. A per-chunk *pending put array* (PPA) maps each thread either to a cell in the chunk that the thread is currently attempting to put into and a corresponding version, or $\langle\perp,\perp\rangle$, indicating that the thread is currently not performing a put. The purpose of the PPA will become evident below.

Rebalance-related data. The structure of the linked list changes only during rebalances; rebalance sometimes needs to mark a chunk's next pointer as immutable, to avoid races around linked list updates. To this end, it uses the mark field. Finally, the chunk holds a rebalance data structure for managing rebalances. It includes a status field that indicates whether the chunk is undergoing rebalancing: an *infant* status means that the chunk is still being initialized; a *normal* chart is part of the data structure; and a *frozen* chunk is one that is undergoing rebalance. The parent and ro fields are used internally by rebalance and are explained in Section 3.3 below.

3.2 KiWi operations

Our algorithm makes use of atomic *compare-and-swap* – CAS(x,old,new), *fetch-and-increment* – F&I(x), and *fetch-and-add* – F&A(x,a) instructions for synchronizing access to shared data; all impose memory fences. A pseudocode of KiWi operations is given in Algorithms 2 and 3. We first describe how scans interact with puts to implement atomic scans. We proceed to describe the put operation and then gets and scans. Finally, we explain how the order (linearization) between operations is determined.

3.2.1 Helping puts. The interaction between put and scan operations is somewhat involved. In a nutshell, a put uses the current value of GV, whereas a scan begins by atomically fetching-and-incrementing GV into a local variable ver, causing all future puts to write larger versions than the fetched one. The scan then uses ver as its scan time, i.e., it returns for each scanned key the latest version that does not exceed ver.

However, a race may arise if put(key,val) reads a global version equal to ver for its data and then stalls while a concurrent scan obtains ver as its scan time and continues to read key before the put writes val for key with ver. In this example, val should be included in the scan (since its version equals the scan time), but it is not (because it occurs late).

We overcome this scenario by having scans *help* pending puts assign versions to values they write. To this end, puts publish their existence in the PPA of the chunk where they write, whereas scans call helpPendingPuts in each chunk where they read, which checks the PPA and helps any relevant pending put threads it encounters (lines 48–53). The helping here is minimal – it consists of a single CAS that assigns a version to the pending put (line 53). For example, in Figure 2, the scan helps put(k1,a) by setting its version to the current global version, namely 8. This orders the put after the scan, so the scan may return the old value.

Since scans use version numbers in order to decide which puts they take into account, the order of put operations is determined by the order of their versions. For consistency, gets also need to rely on version numbers. When a get(key) encounters a pending put(key,v) with no version, it cannot simply ignore the written value, because the put might end up ordered earlier than the get. Gets

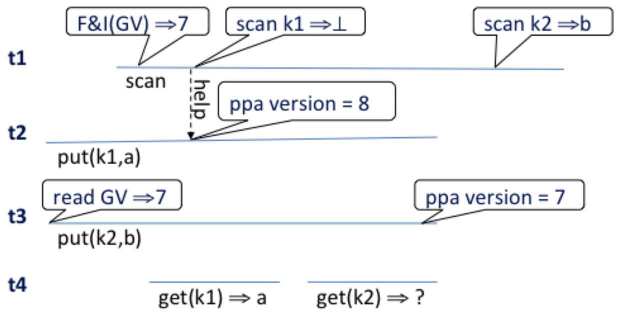


Fig. 2. Example of a scan operation enforcing an order between puts: The scan assigns put(k1,a) a new version (8), whereas put(k2,b) later completes with an old version (7). We see that if get(k2) does not help put(k2,b), the gets see puts in a different order than the scan.

therefore call helpPendingPuts to help pending puts as scans do. This is depicted in Figure 2, where the get must help put(k2,b) obtain a version, because ignoring it would order the gets inconsistently with the version order that would later be observed by the scan.

3.2.2 Put implementation. The put operation appears in Algorithm 2. It consists of three phases: (1) locate the target chunk and prepare a cell to insert with the written value; (2) obtain a version while synchronizing with concurrent scans, gets, and rebalances via the PPA; and (3) connect the new cell to the linked list.

To locate the target chunk (lines 25-32), we first lookup the search key in the index. If the returned chunk is frozen, it is possible that it is no longer connected to the linked list, and hence we re-search the index with a smaller key. Because the head of the linked list is never frozen, and the number of keys is finite, this process eventually returns an unfrozen chunk. Locate then traverses the linked list until the chunk the sought key belongs in.

The first phase of put (lines 2–9) locates the target chunk C and allocates space for the key and the variable-length value by increasing C's array counters to the next available indexes i and j for k and v, resp. This is done using atomic F&I and F&A, so in case of concurrent put operations, each thread gets its own cells.

Before increasing i and j, put checks if rebalancing is needed, because the chunk is full, imbalanced, or immutable, as discussed in the next section. This is done by the procedure checkRebalance given below, which returns false in case no rebalance is needed, and otherwise completes (or restarts) the put. After increasing i and j, put verifies that they are not too large, and if so, proceeds to write values into k[i] and v[j], without a version at this point; note that k[i] is not yet connected to the linked list.

The second phase (lines 10–15) publishes i in the thread's entry in C's PPA, and then uses CAS to set the version to the current value of GV. The CAS may fail in two possible ways. First, if the chunk is undergoing rebalancing, then the reblanacing thread may have set the thread's PPA version to frozen. In this case, the put cannot proceed since the chunk is deemed immutable. Instead, it invokes rebalance, and if rebalance returns false indicating that it did not insert the put's key and value, the put restarts (lines 13–14). (Invoking rebalance on a chunk that is already being rebalanced is done for lock-freedom, given that the original rebalancing thread may be stalled.) Second, a helping thread may have already set the version; to account both for this case and for the case CAS succeeds, put uses the version from the PPA, and copies it to k[i] (line 15).

Algorithm 2 KiWi put operation – pseudocode for thread t.

```
1:  procedure PUT(key, val)
      ▷ 1. prepare cell to insert
2:      $C \leftarrow$ locate(key)
3:      if checkRebalance($C$, key, val) then return      ▷ required rebalance completed the put
4:      j $\leftarrow$ F&A($C$.vCounter, val.size)          ▷ alF place for value
5:      i $\leftarrow$ F&I($C$.kCounter)                    ▷ allocate cell in linked list
6:      if j $\geq C$.v.size $\lor$ i $\geq C$.k.size then
7:          if $\neg$rebalance($C$, key, val) then put(key, val)
8:      v[j] $\leftarrow$ val
9:      k[i] $\leftarrow \langle$key, $\perp$, j, $\perp\rangle$      ▷ version and list connection not set yet
      ▷ 2. set PPA version
10:     $C$.ppa[t].idx $\leftarrow$ i
11:     gv $\leftarrow$ GV
12:     CAS($C$.ppa[t], $\langle\perp$, i $\rangle$, $\langle$ gv, i $\rangle$)
13:     if $C$.ppa[t].ver = frozen then                     ▷ $C$ is being rebalanced
14:         if $\neg$rebalance($C$, key, val) then put(key, val)
15:     $C$.k[i].ver $\leftarrow C$.ppa[t].ver
      ▷ 3. add k[i] to linked list
16:     repeat
17:         c$\leftarrow$ find(key, k[i].ver, $C$)    ▷ search $C$.k (use binary search up to $C$.batchedIndex)
18:         if c = $\perp$ then                                ▷ not found
19:             link $C$.k[i] to the list using CAS
20:             if CAS succeeded then break
21:         else if c.valPtr = j'<j then                      ▷ overwrite
22:             CAS(c.valPtr, j', j)
23:     until c.valPtr $\geq$ j
24:     $C$.ppa[t] $\leftarrow \langle\perp, \perp\rangle$

25: procedure LOCATE(key)
26:     search_key $\leftarrow$ key
27:     repeat                                  ▷ Find unfrozen chunk to begin traversal
28:         $C_0 \leftarrow$ index.lookup(search_key)           ▷ $C_0$.minKey $\leq$ search_key
29:         search_key $\leftarrow C_0$.minKey$-1$      ▷ if $C_0$ is frozen, look for smaller key
30:     until $C_0$ is not frozen
31:     traverse linked list from $C_0$ until the last chunk $C$ s.t. $C$.next = $\perp$ or $C$.next.minKey < key
32:     return $C$
```

The third phase, (lines 16–23), adds k[i] to the linked list. To find the insertion point, it first uses binary search on the batched prefix and then traverses the remaining linked list. If the linked list does not contain a cell with the same key and version, then k[i] is linked to the list (line 19). Otherwise, ties between two puts with the same key and version are broken based on the indexes of their allocated cells in v. If the put that allocates cell j finds in the linked list a cell with index j' with the same key and version such that j'<j, it uses CAS to replace that cell's valPtr to point to v[j] (line 22). If j'>j then the put does nothing, since its value has effectively been overwritten. Note

Algorithm 3 KiWi get and scan operations – pseudocode for thread t.

```
33: procedure GET(key)
34:     $C \leftarrow$ locate(key)
35:     helpPendingPuts($C$, key, key)
36:     return findLatest(key, $\infty$, $C$)

37: procedure SCAN(fromKey, toKey)
        ▷ 1. obtain version - synchronize with rebalance via PSA
38:     psa[t] $\leftarrow \langle$?, seq, fromKey, toKey$\rangle$                ▷ seq is thread-local
39:     ver $\leftarrow$ F&I(GV)
40:     CAS(psa[t], $\langle$?, seq, fromKey, toKey$\rangle$, $\langle$ver, seq, fromKey, toKey$\rangle$)
41:     ver $\leftarrow$ psa[t].ver
        ▷ 2. scan relevant keys
42:     $C \leftarrow$ locate(fromKey)
43:     for each chunk $C$ in query range do
44:         helpPendingPuts($C$, fromKey, toKey)
45:         for each key in query range do
46:             return findLatest(key, ver, $C$)
47:     psa[t] $\leftarrow \langle \perp$, seq++, $\perp, \perp \rangle$

48: procedure HELPPENDINGPUTS($C$, fromKey, toKey)
49:     for each entry $e$ in $C$.ppa do
50:         idx $\leftarrow$ $e$.idx
51:         if $C$.k[idx].key $\in$ [fromKey, toKey] then
52:             gv $\leftarrow$ GV
53:             CAS($e$, $\langle \perp$, idx $\rangle$, $\langle$ gv, idx $\rangle$)

54: procedure FINDLATEST(key, ver, C)
55:     search key in $C$.k and $C$.ppa
56:     if found at least one cell with key and version $\leq$ver then
57:         return one with highest version, break ties by valPtr
58:     return $\perp$
```

that in both cases some cell, either j or j', remains allocated but is not connected to the linked list. Unconnected cells are compacted by the rebalancing process. Finally, the PPA version is cleared (line 24).

3.2.3 Gets and scans. Gets and scans are presented in Algorithm 3. A get(key) begins by locating the chunk C where key belongs (line 34). If there is a pending put to key that does not have a version yet, (i.e., its version is $\perp$), get attempts to help it by setting its version to the current value of GV using CAS (line 53). (CAS may fail in case the put sets its own version or is helped or frozen by another thread). It then calls findLatest() to find the latest version of the searched key.

The findLatest() function (line 54) performs a binary search on the batched prefix, and continues to traverse the in-chunk linked list until it either finds key or finds that it does not exist. In addition, findLatest() checks the PPA for potential pending puts of the target key, ignoring entries with no versions as these were added after the help. In case multiple versions of key exist, it returns the one with the highest version. If a pending put has the same version for the sought key as an entry in the linked list, then the one with the larger valPtr is returned.

A scan first determines its scan time (lines 38–41). It obtains a unique version via F&I of GV, and attempts to set it as its scan time while synchronizing itself relative to helping rebalance operations as described below.

It then reads all the keys in the relevant range (lines 42–46) by locating fromKey, and then traversing the list of chunks until the next chunk's minKey exceeds toKey. Within each chunk, it proceeds as get does to help all pending puts and find the latest version of each key.

	scan	**put**	**rebalance**
scan	F&I GV	–	–
put	CAS ppa[t].ver	by version, then F&A vCounter	–
rebalance	CAS psa[t].ver	CAS to frozen ppa[t].ver	CAS rebalanceObj

Table 2. Atomic operations and rendezvous points determining order between KiWi procedures.

3.2.4 Ordering operations. The order among concurrently executing KiWi procedures is determined by atomic hardware operations (F&I, F&A, or CAS) on pertinent memory locations. Table 2 summarizes the rendezvous points for different types of operations. For brevity, we omit gets from the table.

Each scan has a unique version. The order between concurrent scans is determined by the order in which they (or the rebalance threads that help them) perform F&I on GV. Scans (and gets) order themselves relative to a put by thread t via ppa[t].ver in the chunk where the put occurs.

The order between puts that attempt to insert the same key is determined by their versions, which reflects their order wrt ongoing scans. Puts that have the same version, (i.e., the order between them is not determined by scans), are ordered according to the order in which they succeed to fetch-and-add vCounter. Rebalance operations are discussed below.

3.3 Rebalancing

Section 3.3.1 discusses the life-cycle of a KiWi chunk, and in particular, when it is rebalanced. Section 3.3.2 then walks through the stages of the rebalance process.

3.3.1 Triggering rebalance and chunk life-cycle. We saw that put calls checkRebalance(C) in line 3 before adding a new key to C. This procedure triggers rebalance(C) whenever C is full or otherwise unbalanced, according to some specified *rebalance policy*; we refer to C as the *trigger chunk* of the rebalance. To ensure liveness, it is essential to trigger rebalance only after at least one put – and hopefully many puts – succeed in the chunk. We therefore assume that chunks are created "in good standing", i.e., sorted and no more than half full and the policy isn't overly aggressive in triggering rebalance.

To address the immediate problem for which rebalance is triggered, KiWi could, in principle, restrict itself to the trigger chunk: It can free up space by *compacting* C, i.e., removing deleted values, values that are no longer in the linked list because their keys have been over-written, as well as values that pertain to old versions that are not required by any active scan; if this does not suffice (because all the information in C is needed), KiWi may *split* the chunk. Furthermore, it can address the imbalance by *sorting* the chunk.

Algorithm 4 The checkRebalance procedure.

```
59: procedure checkRebalance($C$, key, val)
60:     if $C$.status=infant then
61:         normalize($C$.parent)
62:         put(key, val) s
63:         return true
64:     if $C$.vCounter $\geq$ $C$.v.size $\vee$ $C$.kCounter $\geq$ $C$.k.size $\vee$ $C$.status = frozen $\vee$ policy($C$) then
65:         if $\neg$rebalance($C$, key, val) then put(key, val)
66:         return true
67:     return false
```

The problem with this approach is that it may leave under-utilized chunks in the data structure forever. KiWi improves space utilization by allowing chunks to *merge*, or more generally, *engaging* a number of old chunks in the rebalance, and replacing all of them with any number of new ones. The chunks to engage are determined by the rebalance policy.

Rebalance clones the relevant data from all engaged chunks into new chunks, and then replaces the engaged chunks with the new ones in the data structure. Cloning creates a window when the same data resides at two chunks – new and old. In order for get and scan to be wait-free, the chunks remain accessible for reading during this period. But in order to avoid inconsistencies, both chunks (old and new) are immutable throughout the window.

This defines a life cycle for chunks: they are created as immutable *infants* by some *parent* trigger chunk C; they become *normal* mutable chunks at the end of the rebalance process; and finally, they become *frozen* (and again immutable) when they are about to be replaced. (We assume that a complementary garbage-collection mechanism eventually removes disconnected frozen chunks.) The chunk's status (infant, normal, or frozen), the pointer to its parent, and the pointer to the appropriate rebalance object are part of the rebalance data stored in the chunk (see Algorithm 1).

The checkRebalance(C) procedure is given in Algorithm 4. It checks whether C is immutable, and if so, helps complete the process that makes it immutable (C's parent in case C is an infant, and C in case it is frozen). The rebalance procedure consists of two functions, rebalance and normalize; in case the chunk's parent is helped only the latter is performed as explained below. In addition, rebalance on C may be triggered in case the chunk is full or if the rebalance policy chooses to do so. Note that put calls checkRebalance before incrementing kCounter and vCounter in order to avoid filling up infant chunks. The rebalance procedure takes the put's key and value as parameters, and attempts to piggyback the put on the rebalance, i.e., insert the key and value to the newly created chunk. In case it fails, it returns false, in which case the put is restarted.

The policy will typically choose to rebalance C whenever C is full or under-utilized, as well as when its batched prefix becomes too small relative to the number of keys in C's linked list. In order to stagger rebalance attempts in case of many insertions to the same chunk, the policy can make probabilistic decisions: If a chunk is nearly full or somewhat under-utilized or unbalanced, then the policy may flip a coin to decide whether to invoke rebalance or not.

3.3.2 *Rebalance stages.* Rebalance proceeds in the following seven stages:

(1) *Engage* – agree on the list of chunks to engage.
(2) *Freeze* – make engaged chunks immutable.
(3) *Pick minimal version* – to keep in compaction.
(4) *Build* – create infant chunks to replace engaged ones.

(5) *Replace* – swap new chunks for old ones in list.
(6) *Update index* – un-index old chunks, index new ones.
(7) *Normalize* – make the new chunks *mutable*.

If the first check of checkRebalance() decides to help rebalance a chunk's parent, then rebalance starts in stage 6, since the chunk's reachability implies that stage 5 is complete. In other cases, (a frozen chunk or a new trigger chunk), rebalance cycles through all seven stages. This is safe because all stages are idempotent, and ensures lock-freedom, namely, progress in case the original rebalance stalls. The first five stages are performed in the rebalance procedure, whereas the last two are performed in normalize. Pseudocode for these operations is given in Algorithms 5 – 7.

1. Engagement. Since multiple threads may simultaneously execute rebalance(C), they need to reach *consensus* regarding the set of engaged chunks. The consensus is managed via pointers from the chunks to a dedicated rebalance object ro. Once a chunk is engaged in a rebalance it cannot be engaged with another rebalance. The engaged chunks in a particular rebalance always form a contiguous sector of the chunks linked list. For simplicity, this sector always starts from the trigger chunk forward, though in principle it is possible to grow the sector backwards from the trigger chunk as well. A rebalance object holds pointers to two chunks, first (the trigger chunk) and next (the next potential chunk to engage in the rebalance). The engagement preserves the following invariant:

Invariant 3.1. *Consider a rebalance object ro. If ro.next≠⊥ then for every chunk C in the linked list from ro.first to the chunk before ro.next, C.ro=ro.*

Engagement begins by agreeing on the ro to use. This is done by creating a rebalance object referring to the trigger chunk C (line 69), attempting to set C.ro to this rebalance object via CAS (line 70), and (3) using the ro in C.ro (line 71). Note that the latter was set by a successful CAS of some rebalance thread. Next, we try to engage ensuing chunks in the list one by one. In each iteration, we consult the policy whether to engage the next chunk. We then use CAS to change ro.next to either ro.next.next (in case we decide to engage another chunk), or ⊥ (indicating that it is time to stop engaging chunks). We exit the loop when ro.next is ⊥, and then set the local variable last to the last engaged chunk.

2. Freezing. Once the list of engaged chunks is finalized, we freeze them so no data will be added to them while they are being cloned. Recall that puts become visible to concurrent retrievals once they publish themselves in the chunk's PPA, and that before doing so, they check if the chunk is frozen. However, we need to account for the scenario where a chunk becomes frozen after put checks its status and before the put publishes itself in PPA. To this end, rebalance traverses all the PPA entries and attempts to set their versions to frozen using CAS. If the CAS is successful, the put will fail to assign itself a version (Algorithm 2, line 13). Otherwise, the put has already assigned its version, and rebalance can take it into account during cloning.

3. Determining the minimal read version and helping scans. We need to clone all data versions that might still be needed by scans. To this end, we compute minVersion, the minimum read point among all active and future scans – this is the smallest version among those published in PSA and the current GV.

Since a scan cannot atomically obtain a scan time from GV and publish it in PSA, rebalance cannot ignore scans that have started but did not publish a version yet. We therefore use a helping mechanism: scan first publishes ? in PSA (Algorithm 3, line 38) indicating its intent to obtain a version, then fetches-and-increments the global version and uses CAS to update the version from ? to the one it obtained.

Algorithm 5 KiWi's rebalance procedure – stages (1)–(3).

```
68: procedure REBALANCE(C, put_key, put_val)
        ▷ 1. engage
69:     tmp ← new rebalance object, with first=C, next=C.next
70:     CAS(C.ro, ⊥, tmp)
71:     ro ← C.ro
72:     last ← C
73:     while ro.next ≠ ⊥ do
74:         next ← ro.next
75:         if policy(next) then                          ▷ try to engage next
76:             CAS(next.ro, ⊥, ro)
77:             if next.ro = ro then                      ▷ engaged next
78:                 CAS(ro.next, next, next.next)
79:                 last ← next
80:             else
81:                 CAS(ro.next, next, ⊥)
82:         else
83:             CAS(ro.next, next, ⊥)
84:     while last.next.ro = ro do     ▷ search for the last concurrently engaged chunk
85:         last ← last.next
        ▷ 2. freeze
86:     for each chunk c from ro.first to last do
87:         c.status ← frozen
88:         for each entry e in c.ppa do
89:             idx ← e.idx
90:             CAS(e, ⟨⊥, idx⟩, ⟨frozen, idx⟩)
        ▷ 3. pick minimal version
91:     minVersion ← GV
92:     for each psa[t] = ⟨ver, seq, from, to⟩ do
93:         if ro.first.minKey ≤ to ∧ last.next.minKey > from then
94:             if ver=? then add ⟨t, seq, from, to⟩ to toHelp
95:             else minVersion ← min(minVersion, ver)
96:     if toHelp ≠ ∅ then
97:         ver ← F&I(GV)
98:         for each ⟨t, seq, from, to⟩ ∈ toHelp do
99:             CAS(psa[t], ⟨?,seq,from,to⟩, ⟨ver,seq,from,to⟩)
100:            minVersion ← min(minVersion, psa[t].ver)
101:    return build(C, put_key, put_val, ro, last, minVersion) ▷ Build new chunks, replace old ones
```

Concurrent rebalance operations help scans install a version in started entries; monotonically increasing counters are used to prevent ABA races where an old rebalance "helps" a new scan. Specifically, rebalance does the following: it scans the PSA for entries with ? whose range overlaps the range covered by the engaged chunks (line 94); if any are found, it helps them (lines 96–100) by fetching-and-incrementing GV and for every psa[t]= ⟨?, . . . ⟩ entry found, attempting to CAS psa[t] to hold the new version. Note that scan's CAS (Algorithm 3, line 40) might fail in case it is helped, but either way, it uses the version written by some successful CAS (line 41).

Algorithm 6 KiWi's rebalance operation – stages (4) build and (5) replace.

```
102: procedure BUILD($C$, put_key, put_val, ro, last, minVersion)
        ▷ 4. build
103:    $C_f \leftarrow C_n \leftarrow$ new chunk with minKey=$C$.minKey, parent=$C$, status=infant
104:    for each chunk $C_o$ from ro.first to last do
105:        if $C_o$.minKey $\leq$ put_key $<$ $C_o$.next.minKey then
106:            toPut $\leftarrow \{\langle$put_key, GV, put_val$\rangle\}$
107:        else
108:            toPut $\leftarrow \emptyset$
109:        for each k in $C_o$.ppa $\cup$ $C_o$.k $\cup$ toPut in ascending order do
110:            if $C_n$ is more than half full then
111:                $C_n$.next $\leftarrow$ new chunk with minKey=k, parent=$C$, status=infant
112:                $C_n \leftarrow C_n$.next
113:            for each version $\langle$ ver, val $\rangle$ of k, in descending order do
114:                if val = $\perp$ then break                              ▷ eliminate tombstones
115:                insert $\langle$ k, ver, val $\rangle$ to $C_n$
116:                if ver $<$ minVersion then break
        ▷ 5. replace
117:    repeat          ▷ set next pointer & mark bit of last old chunk to $C_n$'s next pointer & true
118:        $C_n$.next $\leftarrow$ last.next
119:        CAS(last.next+mark, $C_n$.next+false, $C_n$.next+true)
120:    until last.next+mark = $C_n$.next+true
121:    do
122:        pred $\leftarrow$ $C$'s predecessor
123:        if CAS(pred.next+mark, $C$+false, $C_f$+false) then                   ▷ success
124:            normalize($C$)
125:            return true
126:        if pred.next.parent = $C$ then                          ▷ someone else succeeded
127:            normalize($C$)
128:            return false
129:        rebalance(pred, $\perp$, $\perp$)                    ▷ insertion failed, help predecessor
130:    while true                                                          ▷ and retry
```

4. Creating new chunks and completing the put. The next stage creates new chunks to replace the engaged ones. It uses thread-local variables C_f and C_n to hold pointers to the first and last new chunks, respectively. It traverses the list of engaged chunks from ro.first to last. In each chunk, it collects data both from the intra-chunk linked list and from the PPA. Additionally, the key and value of the put that triggered the rebalance is included in the appropriate chunk. Versions associated with deletions (tombstones) are discarded along with all older versions of the same key. All versions of a key that are older than the last version that does not exceed minVersion can be safely discarded, whereas newer versions are cloned into new chunks. New chunks are created one at a time, as infants, with the trigger chunk as their parent. Keys are added, in sorted order, to a new chunk C_n until it is roughly half full, at which point a new chunk is created and C_n.next is set to this new chunk. (In case the last chunk is too sparse, for example, only a quarter full, it is discarded and its keys are moved to the penultimate chunk). We assume here that the number of versions is much smaller than the chunk size.

5. Data structure update. Next, rebalance attempts to insert the new section into the linked list instead of the engaged one. This involves two steps: First, the next pointer of the tail chunk in the new section needs to take the value of the next pointer in last. Second, the next pointer of the predecessor of ro.first needs to be set to the head of the new chunks' list. In order to execute the two steps atomically, we do the following: We first (lines 117–120) *mark* the next pointer in last as immutable and set the tail of the new chunk sector to its value. We then use CAS to set the next pointer of the predecessor of ro.first (line 123). If CAS succeeds, we return true. If CAS fails because another rebalance (using the same rebalance object) has successfully replaced the trigger chunk with a new one, we simply return false (indicating that the new key and value were not added as part of the rebalance, and hence put should restart) without taking any additional actions. But if CAS fails because some other rebalance had marked the next pointer as immutable (line 120 above), then we recursively help that rebalance complete, and then re-attempt to insert the new chunk sector to the list.

In the special case when the new list is empty (because no data is kept), line 123 CASes the next pointer of the predecessor of ro.first to the next pointer of last.

Algorithm 7 KiWi's rebalance operation – stages (6) update index and (7) normalize.

```
131: procedure NORMALIZE($C$)
132:     discover last by traversing chunks from $C$.ro
133:     discover $C_n$, $C_f$ by traversing from $C$'s predecessor
     ▷ 6. update index
134:     for each chunk $c$ from $C$.ro.first to last do
135:         index.deleteConditional($c$.minKey, $c$)
136:     for each chunk $c$ from $C_f$ to $C_n$ do
137:         do
138:             prev ← index.loadPrev($c$.minKey)
139:             if $c$.frozen then break
140:         while ¬index.putConditional($c$.minKey, prev, $c$)
     ▷ 7. normalize
141:     for each chunk $c$ from $C_f$ to $C_n$ do
142:         CAS($c$.status, infant, normal)
```

6. Index update. The normalize procedure needs to know last, C_n, and C_f in order to delete old chunks from the index and insert new ones. These may be stored in ro by rebalance for this purpose, or may be discovered by traversals.

Since the new chunks are already accessible via the linked list and the old chunks are already frozen, the index update can be lazy, and updates of different chunks can proceed without synchronization. Nevertheless, we need to take into account races with old rebalance operations— a thread that wakes up after being stalled must be prevented from indexing a chunk that had already been supplanted.

To this end, we assume that the index supports a form of semantic load-linked and store-conditional; specifically, it provides the following API: (1) loadPrev(k) — returns the indexed chunk mapped to the highest key that does not exceed k; (2) deleteConditional(k,C) — removes key k only if mapped to chunk C; and (3) putConditional(k,prev,C) — maps k to C provided that the highest key in the index that does not exceed k is mapped to prev. Such an index can be implemented in non-blocking ways using low-level atomic operations [12]; in our implementation, we instead use locks.

To index a new chunk C, we first call loadPrev(C.minKey), then verify that C is not frozen, and if so, add it conditionally to the index. Since chunks are frozen before they are un-indexed, this check ensures that we do not re-index an unindexed chunk. If the conditional put fails and yet the chunk is not frozen, the put is retried. Index removals call deleteConditional(C.minKey,C).

7. Normalization. Finally, the status of the new chunks is set to normal, and put operations may begin to update them. Though it is possible that old (removed) chunks are still being accessed by old get and scan operations at this point, these operations will be ordered before the new puts, so it is acceptable for them to miss the added data. Once all such old operations complete, the old chunks can be reclaimed.

4 CORRECTNESS

In order to lay out foundations for reasoning about correctness, we define in Section 4.1 the model and correctness notions we seek to prove. We proceed to prove the algorithm's safety in Section 4.2 and liveness in Section 4.3.

4.1 Model and Correctness Specification

We consider an asynchronous shared memory model [36] consisting of a collection of shared variables accessed by a finite number of threads, which also have local state. High-level objects, such as a map, are implemented using low-level memory objects supporting atomic read, write, and read-modify-write (e.g., CAS) primitives. Threads *invoke* high-level *operations*, which perform a sequence of *steps* on low-level objects, and finally *return*.

An *algorithm* defines the behaviors of threads executing high-level operations as deterministic state machines, where local state transitions are associated with shared low-level memory accesses (read, write, CAS, etc.) or high-level invocations/responses. A *configuration* describes the local states of all threads and the contents of shared variables. An *initial configuration* is one where all threads and variables are in their initial values. An *execution* of algorithm $\mathcal{A}$ is an alternating sequence of configurations and steps, beginning with some initial configuration, such that configuration transitions occur according to $\mathcal{A}$. Operation op_1 *precedes* operation op_2 in an execution if op_1's return step precedes op_2's invoke step; two operations are *concurrent* in execution σ if neither precedes the other, that is, both are invoked in σ before either returns. In a *sequential* execution, there are no concurrent operations. We use the notion of time t during an execution σ to refer to the configuration reached after the t^{th} step in σ. An *interval* of execution σ is a sub-sequence of σ. The *interval of an operation op* in σ starts with the invocation step of *op* and ends with the configuration following the return from *op* or the end of σ, if there is no such return.

Our correctness notion is *linearizability*, which intuitively means that the object "appears to be" executing sequentially. More formally, the *history* $H(\sigma)$ of execution σ is the sequence of invocations and returns occurring in σ. In a sequential history, each invocation is immediately followed by its return. An object is specified using a *sequential specification*, which is the set of its allowed sequential histories.

For a history h, *complete(h)* is the sequence obtained by removing invocations with no responses from h. We assume that histories are *well-formed*, meaning that the sub-sequence of each thread's steps in a history is sequential. An algorithm is *linearizable* [25] if each of its histories $h = H(\sigma)$

can be extended by adding zero or more response events to a history h', so that *complete(h')* has a sequential permutation that preserves h's precedence relation and satisfies the object's sequential specification. Thus, a linearizable algorithm provides the illusion that each invoked operation takes effect instantaneously at some *linearization point* inside its interval.

For liveness, we consider two notions: *wait-freedom* requires that *every* operation return within a finite number of its own steps, whereas *lock-freedom* requires only that *some* operation return within a finite number of steps. The former is sometimes called *starvation-freedom* and the latter – *non-blocking*.

KiWi implements a linearizable map offering lock-free put operations and wait-free get and scan operations. In its sequential specification, get and scan return the latest value inserted by a put for each key in their ranges.

4.2 KiWi's Linearizability

In Section 4.2.1 we show that the rebalance process preserves the data structure's integrity and contents. We proceed to prove that KiWi is linearizable by identifying, for every operation in a given execution, a linearization point between its invoke and return steps, so that the operation "appears to" occur atomically at this point. We discuss put operations in Section 4.2.2, and gets and scans in Section 4.2.3. The linearization point of operation *op* is denoted LP(*op*).

4.2.1 Rebalance. We first argue that rebalance operations preserve the integrity of the data structure. To this end, we introduce some definitions. We say that a chunk C is *accessible* in KiWi if C is connected to the linked list, that is, if traversing the linked list from its head to its tail goes through C. While a chunk is accessible its key range is well-defined: We say that key k is in the *range* of chunk C if $k \geq C$.minKey and $k < C$.next.minKey.

When all the entries in a chunk's PPA are frozen, we say that the chunk is *frozen*. Observe that a put operation can successfully assign a PPA version (phase 2 of put) in chunk C at a time t only if (1) C is accessible at some time $t' < t$, and (2) C is not frozen at time t. This is because once a thread's PPA entry is frozen, its attempt to CAS it inevitably fails and it triggers rebalance. We say that a chunk is *mutable* if these two conditions are satisfied. Similarly, a chunk is *immutable* before it first becomes accessible and again after the freezing stage of its rebalance is complete.

Rebalance preserves the following invariant:

Invariant 4.1. *At any point in an execution of KiWi, for every key k,*

(1) *the minKey values in the linked list are monotonically increasing (so k is in the range of exactly one accessible chunk);*
(2) *k is in the range of at most one mutable chunk; and*
(3) *querying the index for k returns a chunk C s.t. C.minKey $\leq k$ and C is either accessible or frozen.*

Proof. (1) Observe that when a segment of new chunks is connected instead of a sequence of old ones, C_f.minKey is equal to C.minKey, and the next pointer in C's predecessor is replaced via CAS from C to C_f (line 123) hence the invariant is preserved on the left side of the new segment. The invariant is also preserved on the right side of the new segment because each new chunk's minKey is set to some key encountered in the old segment before last, and C_n.next is guaranteed to be last.next (lines 117–120).

(2) The rebalance protocol does not link new chunks to the list (stage (5)) before freezing the old chunks holding the same key range (stage (2)). Moreover, once a chunk is engaged (stage (1)), it is associated with a unique rebalance object ro whose next pointer is set to $\perp$, and hence the segment of chunks associated with ro cannot change. Using a CAS to set C's predecessor's

next pointer to C_f ensures that the old immutable chunk sequence is replaced by at most one new accessible mutable chunk sequence.

(3) Chunks are indexed according to their minKey, and the rebalance protocol does not index new chunks (stage (6)) before making them accessible (stage (5)). Before a chunk ceases to be accessible, it must be frozen.

□

From Invariant 4.1, we derive the correct execution of the locate function:

Corollary 4.2. *The locate(key) function (lines 25-32) always returns a chunk C s.t. key is in C's range. Moreover, if key is in the range of some chunk C that is mutable throughout the execution of locate, then locate returns C.*

In addition to preserving the data structure's integrity, rebalance ensures that no key-value pairs disappear from the data structure due to rebalancing. We say that a key-value pair ⟨key, val⟩ is *stored in* KiWi at time t in execution σ if invoking get(key) at the end of σ and allowing it to complete without interfering steps of other threads returns val. We show the following:

Proposition 4.3. *If ⟨key, val⟩ is* stored in *KiWi at time t in execution σ and no subsequent put(key,_) operations are invoked in σ, then ⟨key, val⟩ is* stored in *KiWi at all times $t' > t$ in σ.*

Proof. By Invariant 4.1(1), key is in the range of exactly one accessible chunk C at time t, which get(key) locates, and the returned val is the one associated with key with the highest version (with ties broken by valPtr). Observe that as long as C remains accessible at time t', its range does not change because minKey is invariant, and if C's successor is replaced by rebalance, it is replaced with a chunk with the same minKey. Since no subsequent put(key,_) operations are invoked in σ, val remains the highest-version value associated with key in C, and we are done.

It remains to show that a rebalance that removes C does not remove ⟨key, val⟩ from KiWi, from which the proposition follows inductively. This follows from the facts that (1) the highest-versioned value associated with each key in an old chunk C is cloned into a new chunk; and (2) the entire chunk segment is replaced atomically by marking the next pointer of the last engaged chunk to prevent it from changing, and then CASing the predecessor of the first engaged chunk. □

4.2.2 Puts. Puts in a chunk C are ordered (lexicographically) according to their version-value-index pairs $\langle v, j\rangle$, where $\langle v, i\rangle$ is published in the appropriate PPA entry in phase 2 of the put, and C.k[i].valPtr= j; this pair is called the *full version* of the put. We note that in each chunk, the full versions are unique, because threads obtain j using F&A (line 4). First, i is published in ppa[t].idx (line 10) and then the pair gets its final value by a successful CAS of ppa[t].ver, either by the put (line 12) or by a helping thread (line 53). We refer to a step publishing i in ppa[t].idx and to the step executing the successful CAS as the put's *publish time* and the *full version assignment time*, resp., and say that the put *assigns* full version $\langle v, j\rangle$ for its key in C.

We note that each put assigns a full version at most once. Furthermore, as noted above, a full version can only be in a mutable chunk. Once a put operation *po* for key k assigns its full version in chunk C at time t, we can define its linearization point. There are two options:

(1) If at time t *po*'s full version $\langle v, j\rangle$ is the highest for k in C, (among entries in C's PPA and linked list), then LP(*po*) is the last step (by any thread) that reads v from GV before t (line 11 or 52).

(2) Otherwise, let *po*′ be the put(k, _) operation that assigns for k in C the smallest full version exceeding *po*'s before time t. Then LP(*po*) is recursively defined to be at the same time as LP(*po*′). Note that *po*'s full version assignment time exceeds that of *po*′, so the recursive

definition does not induce cycles. In case multiple puts are assigned linearization points the same time, they are linearized in increasing full version order.

By Invariant 4.1, rebalance operations divide puts of key k into disjoint groups; one group per mutable epoch of each chunk covering the key. The following lemma establishes the order among linearization points of puts within one epoch.

Lemma 4.4. *Consider chunk C, key k in the range of C, and an operation po =put(k,_) that assigns $\langle v, j\rangle$ to C.ppa at time t. Then*

(1) *LP(po) is after po allocates location j for its value and before t.*
(2) *LP(po) is a read step of GV that occurs after GV is set to v.*
(3) *LP(po) is after some operation po′ (possibly po, but not necessarily) publishes a put of k in C, where later po′ assigns a full version equal to or greater than $\langle v, j\rangle$.*
(4) *The linearization points of all operations that publish puts of k in C preserve their full version order.*

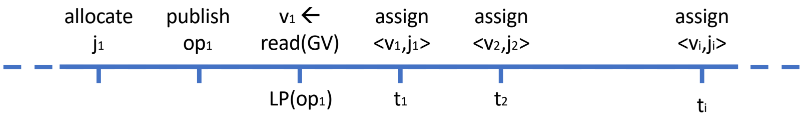
Proof. We consider an execution interval π which spans the execution intervals of all operations that publish k to C . Denote by $t_1, t_2, \ldots$ the finite sequence of times these operations assign versions $v_1, v_2, \ldots$ into entries in the ppa by their order in π, where t_i is the assignment time of operation op_i; denote the locations these operations allocate for their values as $j_1, j_2, \ldots$, respectively (see Figure 3).

The proof is by induction on i. For the base case, we consider op_1. It is the first to publish its version in the ppa (see Figure 3a). All entries of key k from previous epochs were written in C during the rebalance operation that created C; their versions are smaller or equal to v_1 and their locations are smaller than j_1. Therefore, the full versions of these entries are smaller than $\langle v_1, j_1\rangle$ and op_1 is linearized in the last read step of the global version counter that returns v_1 before t_1. Clearly this is after op_1 is published (line 9), and specifically after j_1 is allocated (line 4), and the lemma holds.

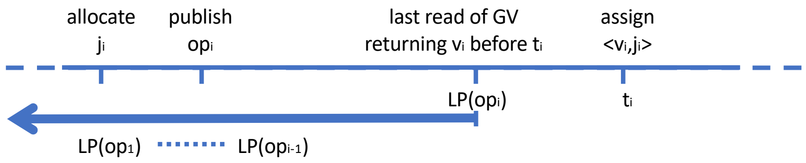
For the induction step, assume the lemma holds for operations $op_1, \ldots op_{i-1}$. We prove the lemma for operation op_i by case analysis. If op_i's full version $\langle v_i, j_i\rangle$ is maximal in C (with respect to all linked cells and published entries with the same key) then LP(op_i) is the last read retrieving v_i from the global counter (see Figure 3b). This step is done after the put is published in the ppa (which is after j_i is allocated) and before t_i, and Conditions 1-3 of the lemma hold. In addition, by the induction hypothesis, linearization points of $op_1, \ldots op_{i-1}$ preserve their full version order. They are all linearized in read steps after the global version counter is set to their versions, specifically not later than LP(op_i)—the latest read step returning the maximal version, hence Condition 4 holds as well.

Otherwise, another operation op_l published $\langle v_l, j_l\rangle$ in t_l before t_i, s.t. $\langle v_l, j_l\rangle > \langle v_i, j_i\rangle$. By definition, op_i is linearized exactly at the point (LP(op_l)) which preserves the full version order of the operations, and Condition 4 holds. By Condition 3 of the induction hypothesis, LP(op_l) is after an operation publishes k to C (and later assigns a full version equal or greater than $\langle v_l, j_l\rangle$). Since $\langle v_l, j_l\rangle > \langle v_i, j_i\rangle$, Condition 3 also holds.

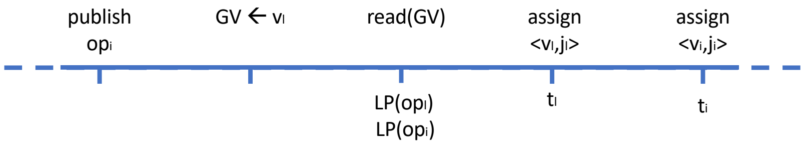
It is left to discuss Conditions 1 and 2. Consider first the case where $v_l > v_i$ (see Figure 3c). By Condition 2 of the induction hypothesis, LP(op_l) is in a read step of the global version counter that occurs after it is set to v_l. Eventually, op_i assigns version v_i, which is smaller than v_l. This implies that op_i publishes the operation in the ppa before the version counter is set to v_l, and LP(op_i) satisfies Conditions 1 and 2. If $v_l = v_i$ and $j_l > j_i$ (see Figure 3d) then op_l allocates j_l after op_i allocates j_i. By Condition 1 of the induction hypothesis, LP(op_l) is after op_l allocates j_l and before t_l, and LP(op_i) satisfies Conditions 1 and 2.



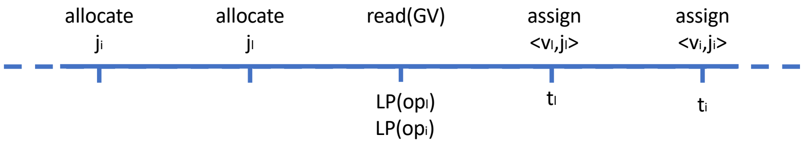
(a) Base case: $\langle v_1, j_1 \rangle$ is maximal in t_1



(b) Induction step: case $\langle v_i, j_i \rangle$ is maximal in t_j



(c) Induction step: case $v_l > v_i$ and $t_l < t_i$



(d) Induction step: case $v_l = v_i, j_l > j_i$ and $t_l < t_i$

Fig. 3. Linearization points of put operations publishing key k in chunk C. $t_1, t_2, \ldots$ is the sequence of times these operations assign versions into C's ppa.

□

4.2.3 *Gets and scans.*

The most subtle linearization is of get operations. A get operation *go* may land in a mutable or immutable chunk. We need to linearize *go* before all concurrent puts that *go* misses while seeking the value. For a get operation *go* for a key k in the range of chunk C, we consider the time t when *go* starts traversing C's PPA (line 49 in helpPendingPuts). There are three options:

(1) If C is not accessible from the chunks list at time t, then LP(*go*) is the last step in which C is still accessible from the chunks list.
(2) Else, if *go* does not find k in C then LP(*go*) is at time t.
(3) Else, let *po* be the put operation that inserts the value returned by *go*. LP(*go*) is the latest between time t and immediately after LP(*po*).

The next lemma shows that in the third case no other put writing to k is linearized after LP(*po*) and before LP(*go*).

Lemma 4.5. *Consider a get operation go retrieving the value of key k from chunk C. Let t be the step in which go starts traversing C's PPA. Then:*

(1) *If go does not find k in C, then for each operation po publishing k in C, LP(po) is after t.*

(2) *If go returns the value written by operation po, then LP(go) is after LP(po), and for each $po' \neq po$ publishing k in C, LP(po') is either before LP(po) or after t.*

Proof. First, assume *go* does not find the key in C. It can be inferred from findLatest that all operations publishing k in C are published in the ppa after t. Otherwise, *go* should have observed one of them either in the ppa or in the linked list. By Condition 3 of Lemma 4.4, all operations publishing k in C are linearized after t, and Condition 1 holds.

Next, assume *go* returns the value written by operation po_l, where the full version of po_l is $\langle v_l, j_l \rangle$.

If C is not accessible from the chunks list at time t, then LP(*go*) is in the last step in which C is accessible from the chunks list. Since chunks cease to be mutable before they cease to be accessible, no put operation can publish in C after t, and by Condition 1 of Lemma 4.4 LP(*go*) is after LP(po_l). Otherwise, by definition LP(*go*) is (immediately) after LP(po_l).

Consider an operation $po_m \neq po_l$ publishing k to C with full version $\langle v_m, j_m \rangle$. By Condition 4 of Lemma 4.4, if $\langle v_m, j_m \rangle < \langle v_l, j_l \rangle$ then LP(po_m) is before LP(po_l). It is left to show that if $\langle v_m, j_m \rangle > \langle v_l, j_l \rangle$ then LP(po_m) is after t. It can be inferred from findLatest that all operations publishing k in C with full version $\langle v_m, j_m \rangle > \langle v_l, j_l \rangle$ are published in the ppa after t. Otherwise, *go* should have observed one of them either in the ppa or in the linked list. Condition 3 of Lemma 4.4 implies that these operations are linearized after at least one of them is published in the ppa, hence Condition 2 also holds. □

Scans are linearized when GV is increased beyond their read point, typically by their own F&I, and sometimes by a helping rebalance. Lemma 4.4 is used to prove the following:

Lemma 4.6. *Consider a scan operation so that acquires version v as its read point. For each key k in the range of the scan, so returns the value of the put operation writing to k that is linearized last before LP(so).*

Proof. Consider a put operation *po* that writes to a key in the range of the scan but is not observed by *op*. First, assume *po* acquires version that is less than v. Then it acquires a version before the scan increases the global version counter. Since *so* does not observe *po* in the ppa, *po* completes before *so* reads the entry in the ppa. Moreover, *so* does not observe *po* in the linked list. It can be inferred from the code that another put operation po' with a higher version than *po*'s full version updates the same entry in k in-place. By Condition 4 of Lemma 4.4, *po* is linearized before po' writing the value returned by the scan.

Finally, by Condition 2 of Lemma 4.4 all put operations that are not observed by *so* and acquire versions greater than v are linearized after LP(*so*). □

The definition of the linearization points of scans and get operations imply that these operations are linearized between their invocation and return. Condition 1 of Lemma 4.4 implies the same for puts. Corollary 4.2 shows that gets and scans land in chunks that contain the sought keys in their ranges. Combined with the rebalancing invariants, Lemma 4.5 shows that get operations satisfy their sequential specification, and Lemma 4.6 proves that scans satisfy their sequential specification. Hence we conclude that KiWi implements a linearziable map.

4.3 Liveness

We now prove KiWi's liveness properties. We show that gets and scans are *wait-free*, namely, in any execution, every operation completes within a finite number of steps by its invoking thread. This is proven by showing that the number of iterations in the loops in these procedures is finite. Put operations satisfy a weaker liveness property – *lock-freedom*, namely, in every execution, *some* put operation completes. To prove this, we show that although a put operation can execute an infinite number of rebalances, this can only occur because some other operation (and in fact many operations) successfully complete a put.

We begin by showing that the locate function is wait-free.

Proposition 4.7. *The locate function is wait-free.*

Proof. The function consists of two loops. The first loop executes a finite number of iterations because index.lookup(key) always returns a chunk C_0 with C_0.minKey $\leq$ key, and so the searched keys are monotonically decreasing. Since the number of keys is finite and the head of the list is never frozen, the loop returns an unfrozen chunk after a finite number of iterations. The ensuing traversal is also finite because the linked list is finite. □

Lemma 4.8. *The get and scan functions are wait-free.*

Proof. By Proposition 4.7, the locate function is wait-free. To complete the proof, observe that the loops performed in helpPendingPuts and findLatest iterate (or search) over finites sets – $C.k$ and $C.ppa$, and the scan loop (line 43) traverses chunks in the (finite) linked list. □

Lemma 4.9. *The put function is lock-free.*

Proof. We first show that as long as no rebalance occurs, put completes within a finite number of steps. Beyond rebalance calls and recursive calls to put when it fails, put executes two loops – one in the locate function and one in phase 3 (lines 16–23). The former is wait-free by Proposition 4.7. Next, consider the loop in phase 3 of the put. Observe that the find function is wait-free because it searches a finite array (C.k). Moreover, the CAS in line 19 fails at most a finite number of times (again, due to the finality of C.k), and because we break when it succeeds, it is attempted a finite number of times. Additionally, every time the CAS in line 22 is attempted (and either succeeds or fails), c.valPtr increases, implying that after a finite number of attempts, c.valPtr $\geq$ j, and the loops completes.

Next, consider rebalance triggered by the put. This can occur in checkRebalance (called in line 3), or when either $C.k$ or $C.v$ is full (line 6), or when the chunk is already frozen (line 14). To prove lock-freedom, we will show that every time rebalance returns false (leading to a recursive call to put) or iterates an additional time in an unbounded loop, or recursively calls rebalance, this is because some new put returned successfully. Note in particular, that if a rebalance operation succeeds to replace the engaged sequence (and returns true), then it completes the put that invoked it, and so we do not need to consider this case.

We first argue that with the exception of rebalance stage 5 (replace), all other rebalance stages always complete within a finite number of steps. Because the chunks list is finite, the two loops in stage 1 (engage) are finite. Similarly, the list of engaged chunks and the number of PPA entries in each of them are finite, and so is the nested loop in stage 2 (freeze). In stage 3 (pick minimal version), the loop over the PSA is finite, and it constructs a finite toHelp set, and so the loop over toHelp is finite as well. Likewise, the loops in stage 4 iterate over finite sets, namely, engaged chunks and keys in the data structures therein. In stage 6, the conditional insertion to the index fails only if another thread succeeds to insert a chunk with the same minKey. Because the insertion is retried only as long as the chunk is not frozen, interfering insertions to the index must be due older rebalances that began before the current rebalance, of which there is a finite number. Clearly, stage 7, which performs a single CAS, is also wait-free. Hence, it remains to consider stage 5.

The first loop in stage 5 terminates once the CAS – either of the thread executing the loop or of another thread – succeeds to mark last.next as immutable. The CAS fails if either last.next changes

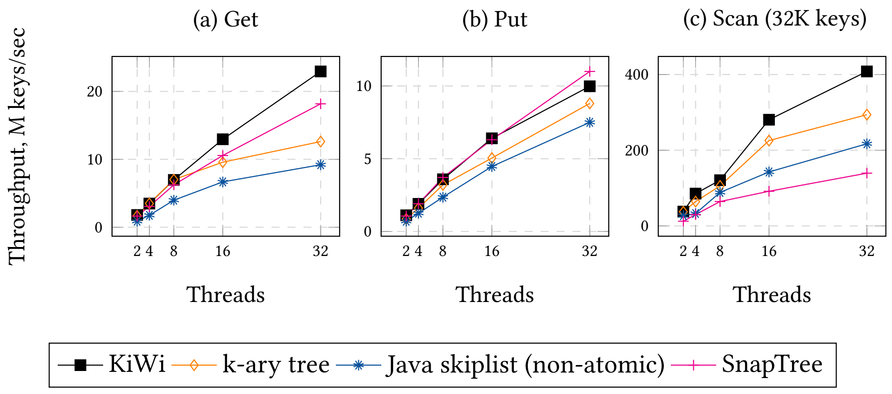


Fig. 4. Throughput scalability with uniform workloads, 1M dataset. (a) Get operations, (b) Put operations, (c) Scan operations.

or another thread marks it. But last.next only changes if some rebalance succceds to replace an engaged chunk succeeding it, and this rebalance returns true upon completion, thus completing a put when it returns.

The second loop returns whenever either the current thread or another thread successfully replaces the next pointer at C's predecessor to a node whose parent is C (C_f in case it is the current thread). They fail to do so only in case the predecessor's next pointer is marked, which in turn only occurs if the predecessor itself is engaged in another rebalance. In this case, the current thread makes a recursive call to rebalance in order to help the predecessor (line 129), and then executes another iteration of the loop. Note that since the linked list is finite, the depth of the recursion is finite, so eventually the inner-most nested recursive call succesfully returns. Once a nested call in line 129 returns after having replaced pred with a new chunk pred', either some thread will succeed to CAS the new predecessor's next pointer causing the calling thread to exit the loop in the next iteration, or the new chunk pred' will undergo rebalance. However, for the latter to occur, it must be the case that at least one new put has completed in pred' following its rebalance. This is because when rebalance completes, pred'.k is sorted and pred'.k and pred'.v are at most half full, and because they hold more entries than twice the number of threads, it is impossible for either of them to fill up due to pending put operations. Thus, rebalance can only be required after a successful put in pred'.

□

5 EVALUATION

5.1 Setup

Implementation. We implement KiWi in Java, using Doug Lea's concurrent skip-list implementation [6] for the index with added locks to support conditional updates. The code makes extensive use of efficient array copy methods [5]. KiWi's chunk size is set to 1024.

The rebalance policy is tuned as follows: checkRebalance invokes rebalance with probability 0.15 whenever the batched prefix consists of less than 0.625 of the linked list. Rebalance engages the next chunk whenever doing so will reduce the number of chunks in the list. We did not implement the piggybacking of puts on rebalance, and instead restart the put after every rebalance.

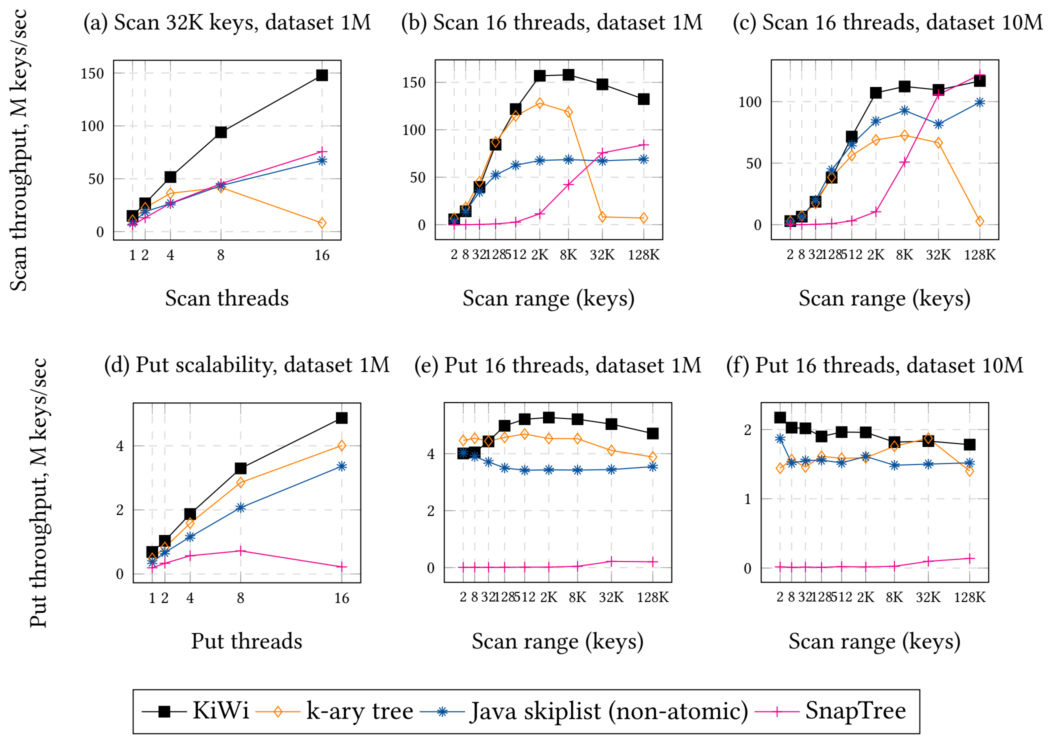


Fig. 5. Throughput scalability with concurrent scans and puts. (a,b) Scan operations, 1M dataset. (c) Scan operations, 10M dataset. (d,e) Put operations, 1M dataset. (f) Put operations, 10M dataset.

Methodology. We leverage the popular *synchrobench* microbenchmark [21] to exercise a variety of workloads. The hardware platform features four Intel Xeon E5-4650 8-core CPUs. Every experiment starts with 20 seconds of *warmup* – inserts and deletes of random keys – to let the HotSpot compiler optimizations take effect. It then runs 10 iterations, 5 seconds each, and averages the results. An iteration fills the map with random (integer, integer) pairs, then exercises some workload. Most experiments start from 1M pairs, except those focusing on high scalability that start from 10M.

Competition. We compare KiWi to Java implementations of three concurrent KV-maps: (1) the traditional skip-list [6] which does not provide linearizable scan semantics, (2) SnapTree[14][2], and (3) k-ary tree [15][3]. For the latter, we use the optimal configuration described in [15] with $k = 64$.

5.2 Results

Basic scenarios: get, put, and scan. We first focus on three simple workloads: (1) get-only (random reads), (2) put-only (random writes, half inserts/updates and half deletes), and (3) scan-only (sequential reads of 32K keys, each starting from a random lower bound).

Figure 4 depicts throughput scalability with the number of worker threads. In get-only scenarios (Figure 4(a)), KiWi outperforms the other algorithms by 1.25x to 2.5x. We explain this by the NUMA-

[2]https://github.com/nbronson/snaptree.
[3]http://www.cs.toronto.edu/~tabrown/kstrq/LockFreeKSTRQ.java.

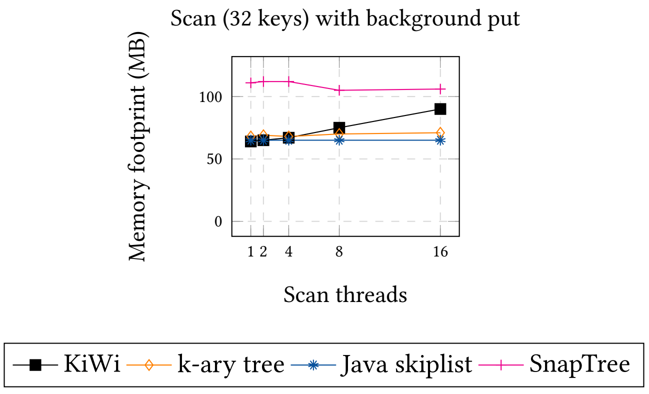


Fig. 6. RAM use with parallel scans and puts, 1M dataset.

and cache-friendly locality of access in its intra-chunk binary search. Under put-only workloads (Figure 4(b)), it also performs well, thanks to avoiding version manipulation. SnapTree, which is optimized for random writes, is approximately 10% faster than KiWi with 32 threads. Note that in general, KiWi's gets are faster than its puts because the latter occasionally incur rebalancing.

KiWi excels in scan performance (Figure 4(c)). For example, with 32 threads, it exceeds its closest competitor, k-ary tree, by over 40%. Here too, KiWi's advantage stems from high locality of access while scanning big chunks.

Concurrent scans and puts. We now turn to the scenario that combines analytics (scan operations) with real-time updates (put operations). This is the primary use case that motivated the design principles behind KiWi. Half of the threads perform scans, whereas the second half performs puts.

Figure 5(a) depicts scan throughput scalability with the number of threads while scanning ranges of 32K keys. Figure 5(b) depicts the throughput for 16 scan threads with varying range sizes. Note that for long scans, k-ary tree's performance deteriorates under contention. This happens because k-ary tree restarts the scan every time a put conflicts with it – i.e., puts make progress but scans get starved. For large ranges, SnapTree has the second-fastest scans because it shared-locks the scanned ranges in advance and iterates unobstructed. Note that KiWi's throughput slightly decreases when the range is particularly big because it takes longer to collect redundant versions, and therefore the scan has to sift through more data. Figure 5(c) depicts similar phenomena for a 10M-key dataset. SnapTree's competitive scan performance comes at the expense of puts, since its locking approach starves concurrent updates. Figures 5(d-f) illustrate this behavior – the latter for a 10M-key dataset.

We study the memory footprints of the solutions in this scenario. We focus on 32-key scans – a setting in which the throughput achieved by all the algorithms except SnapTree is similar. Figure 6 depicts the JVM memory-in-use metric immediately after a full garbage collection that cleans up all the unused objects, averaged across 50 data points. KiWi is on par with k-ary tree and the Java skiplist except with maximal parallelism (16 put threads), in which it consumes 20% more RAM due to intensive version management.

Ordered workload. As a balanced data structure, KiWi provides good performance on non-random workloads. We experiment with a monotonically ordered stream of keys. KiWi achieves a throughput similar to the previous experiments. In contrast, k-ary tree's maximal put throughput in this setting is 730 times slower – approximately 13.6K operations/sec vs KiWi's 9.98M.

6 DISCUSSION

We presented KiWi, a KV-map tailored for real-time analytics applications. KiWi is the first concurrent KV-map to support high-performance atomic scans simultaneously with real-time updates of the data. In contrast to traditional approaches, KiWi shifts the synchronization overhead from puts to scans, and offers lock-free puts and wait-free gets and scans. We demonstrated KiWi's significant performance gains over state-of-the-art KV-map implementations that support atomic scans.

REFERENCES

[1] http://flurrymobile.tumblr.com/post/144245637325/appmatrix.
[2] http://flurrymobile.tumblr.com/post/117769261810/the-phablet-revolution.
[3] Apache HBase – a Distributed Hadoop Database. https://hbase.apache.org/.
[4] Flurry analytics. https://developer.yahoo.com/flurry/docs/analytics/.
[5] Java Array Copy. https://docs.oracle.com/javase/7/docs/api/java/lang/System.html.
[6] Java Concurrent Skip List. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.
[7] A fast and lightweight key/value database library by google. http://code.google.com/p/leveldb, Jan. 2014.
[8] A persistent key-value store for fast storage environments. http://rocksdb.org/, June 2014.
[9] M. Arbel, G. Golan-Gueta, E. Hillel, and I. Keidar. Towards automatic lock removal for scalable synchronization. In *DISC*, pages 170–184, 2015.
[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[11] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *ICDCN*, pages 107–118, 2011.
[12] A. Braginsky and E. Petrank. A lock-free B+tree. In *SPAA*, pages 58–67, 2012.
[13] A. Braginsky, E. Petrank, and N. Cohen. CBPQ: High performance lock-free priority queue. In *Euro-Par*, 2016.
[14] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPOPP*, pages 257–268, 2010.
[15] T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In *OPODIS*, pages 31–45, 2012.
[16] T. Brown and J. Helga. Non-blocking k-ary search trees. In *OPODIS*, pages 207–221, 2011.
[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
[18] B. Chatterjee. Lock-free linearizable 1-dimensional range queries. In *WTTM*, 2016.
[19] K. Fraser. Practical lock-freedom. In *PhD dissertation, University of Cambridge*, 2004.
[20] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *EuroSys*, pages 32:1–32:14, 2015.
[21] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, 2015.
[22] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, pages 206–215, 2004.
[23] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, pages 124–138, 2007.
[24] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
[25] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
[26] I. Keidar and D. Perelman. Multi-versioning in transactional memory. In *Transactional Memory; Foundations, Algorithms, Tools, and Applications*, volume 8913, chapter 7, pages 150–165. 2015.
[27] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *PPoPP*, pages 141–150, 2012.
[28] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.
[29] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, 2012.
[30] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328, 2014.
[31] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *DISC*, pages 224–238, 2013.
[32] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *PPoPP*, pages 151–160, 2012.
[33] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[34] B. Sowell, W. Golab, and M. A. Shah. Minuet: A scalable distributed multiversion b-tree. *Proc. VLDB Endow.*, 5(9):884–895, May 2012.

[35] A. Spiegelman, G. Golan-Gueta, and I. Keidar. Transactional data structure libraries. In *PLDI*, pages 682–696, 2016.

[36] J. L. Welch and H. Attiya. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, 2004.