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

Dmitry Basin* Edward Bortnikov* Anastasia Braginsky* Guy Golan-Gueta†
Eshcar Hillel* Idit Keidar‡* Moshe Sulamy§

*Yahoo Research, Israel †VMWare Research, Israel ‡Technion, Israel §Tel Aviv University
{dbasin,ebortnik,anastas,eshcar}@yahoo-inc.com ggolangueta@vmware.com
idish@ee.technion.ac.il moshe.sulamy@cs.tau.ac.il

Abstract

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.

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 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*.

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

PPoPP '17,, February 04-08, 2017, Austin, TX, USA.

http://dx.doi.org/10.1145/3018743.3018761

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 strucutre, 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 discuss its correctness in Section 5.

Evaluation results. KiWi's Java implementation is available in github[1]. In Section 6 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].

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].

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 unabalnced; 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 Google's Bigtable [17], Apache HBase [3], and others [7, 8]. These technologies combine on-disk indices for persistence with an in-memory KV-map for real-time data acquisition. They often support atomic scans, which 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.

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

	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.

Algorithm 1 KiWi chunk data structure.

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

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 $\perp$ 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 stucture 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 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. 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

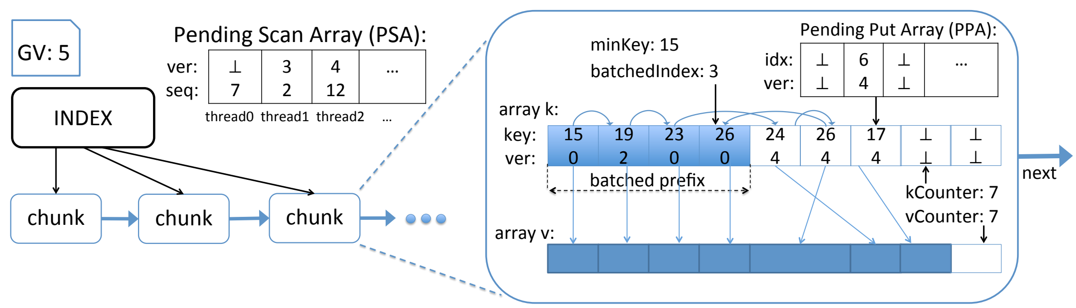

Figure 1: KiWi data structure layout. In the zoomed in chunk (on the right), a pending put by the second thread is attempting to add k[6] to the linked list with key 17 and version 4.

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.

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 Algorithm 2.

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, causing all future puts to write larger versions than the fetched one. Scan then uses the fetched version, 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 41–46). The helping here is minimal – it consists of a single CAS that assigns a version to the pending put (line 46). 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 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.

Put implementation. The put operation appears in the left column of 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 syncrhonizing with

Algorithm 2 KiWi operations – pseudocode for thread t.

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

27: procedure GET(key)
28:   locate target chunk C
29:   helpPendingPuts(C, key, key)
30:   return findLatest(key, ∞, C)

31: procedure SCAN(fromKey, toKey)
      ▷1. obtain version - synchronize with rebalance via PSA
32:   psa[t] ← ⟨?, seq, fromKey, toKey⟩   ▷seq is thread-local
33:   ver ← F&I(GV)
34:   CAS(psa[t], ⟨?, seq, fromKey, toKey⟩, ⟨ver, seq, fromKey, toKey⟩)
35:   ver ← psa[t].ver
      ▷2. scan relevant keys
36:   for each chunk C in query range do
37:       helpPendingPuts(C, fromKey, toKey)
38:       for each key in query range do
39:           return findLatest(key, ver, C)
40:   psa[t] ← ⟨⊥, seq++, ⊥, ⊥⟩

41: procedure HELPPENDINGPUTS(C, fromKey, toKey)
42:   for each entry e in C.ppa do
43:       idx ← e.idx
44:       if C.k[idx].key ∈ [fromKey, toKey] then
45:           gv ← GV
46:           CAS(e, ⟨⊥, idx ⟩, ⟨ gv, idx ⟩)

47: procedure FINDLATEST(key, ver, C)
48:   search key in C.k and C.ppa
49:   if found at least one cell with key and version ≤ver then
50:       return one with highest version, break ties by valPtr
51:   return ⊥
```

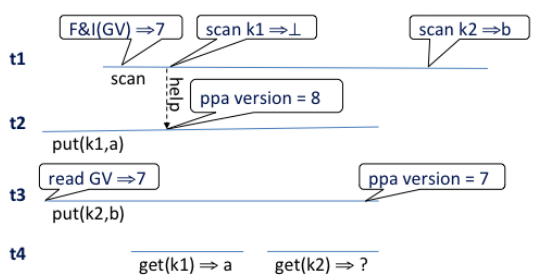

Figure 2: Example of scan operation enforcing order between puts: The scan assigns put(k1,a) a new version (8), whereas put(k2,b) later completes with an old version (7). We see that if get(k2) does not help put(k2,b), the gets see puts in a different order than the scan.

concurrent scans, gets, and rebalances via the PPA; and (3) connect the new cell to the linked list.

The first phase (lines 2–10) locates the target chunk C, traversing the index and the linked list if needed. Later this phase allocates space for the key and the variable-length value, by increasing C's array counters to the next available indices 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 11–16) 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 14–15). (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 16).

The third phase, (lines 17–25), adds k[i] to the linked list. To find the insertion point, it first uses binary search on the

	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 rendevouz points determining order between KiWi procedures.

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 21). Otherwise, ties between two puts with the same key and version are broken based on the indices 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 24). If j'>j then the put does nothing, since its value has effectively been overwritten. Note 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 26).

Gets and scans. Gets and scans are presented in the right-hand column in Algorithm 2. A get(key) begins by querying the index for key, and then (if needed), traverses the linked-list of chunks until the next chunk's minKey exceeds key (line 28). 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 46). (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 47) 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 32–35). 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 36–39) by traversing the list of chunks, and within each chunk, proceeding as get does to help all pending puts and find the latest version of each key.

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 rendevouz 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 of Algorithm 2 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 address the immediate problem, 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.

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 even-

Algorithm 3 The checkRebalance procedure.

```
procedure CHECKREBALANCE(C, key, val)
    if C.status=infant then
        normalize(C.parent)
        put(key, val)
        return true
    if C.vCounter ≥ C.v.size ∨ C.kCounter ≥ C.k.size ∨
        C.status = frozen ∨ policy(C)  then
        if ¬rebalance(C, key, val) then put(key, val)
        return true
    return false
```

tually removes disconnected frozen chunks.) The chunk's status (infant, normal, or frozen), the pointer to parent, and the pointer to rebalance object are part of the rebalance data stored in the chunk (see Algorithm 1).

The checkRebalance(C) procedure is given in Algorithm 3. 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 perfromed as explained below. In addition, if the chunk is full or if the rebalance policy chooses to do so, it also triggers rebalance on C. Note that put calls checkRebalance before incrementing kCounter and vCounter in order to avoid filling up infant chunks. The reblance procedure takes the put's key and value as parameters, and attempts to piggyback the put on the reblanace, 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* – unindex 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 perfomend in the rebalance procedure, whereas the last two are performed in normalize. Pseudocode for these operations is given in Algorithm 4. The two procedures make use of thread-local variables C_f, C_n, and last. The latter tracks the last chunk engaged in rebalancing, whereas the first two hold pointers to the first and last new chunks, respectively.

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 1. *Consider a rebalance object ro. If ro.next$\neq\perp$ 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 (1) creating a rebalance object referring to the trigger chunk C, (2) attempting to set C.ro to the new rebalance object via CAS, and (3) using the ro in C.ro. 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, or $\perp$ indicating that it is time to stop engaging chunks. We exit the loop when ro.next is $\perp$, 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 PPA's 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 14). 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

Algorithm 4 KiWi's rebalance operation.

```
procedure REBALANCE(C, put_key, put_val)
  ▷1. engage
    tmp ← new rebalance object, with first=C, next=C.next
    CAS(C.ro, ⊥, tmp)
    ro ← C.ro
    last ← C
    while ro.next ≠ ⊥ do
        next ← ro.next
        if policy(next) then                              ▷try to engage next
            CAS(next.ro, ⊥, ro)
            if next.ro = ro then                          ▷engaged next
                CAS(ro.next, next, next.next)
                last ← next
            else
                CAS(ro.next, next, ⊥)
        else
            CAS(ro.next, next, ⊥)
  ▷search for the last concurrently engaged chunk
    while last.next.ro = ro do
        last ← last.next
  ▷2. freeze
    for each chunk c from ro.first to last do
        c.status ← frozen
        for each entry e in c.ppa do
            idx ← e.idx
            CAS(e, ⟨⊥, idx⟩, ⟨frozen, idx⟩)
  ▷3. pick minimal version
    minVersion ← GV
    for each psa[t] = ⟨ver, seq, from, to⟩ do
        if ro.first.minKey ≤ to ∧
            last.next.minKey > from then
            if ver=? then add ⟨t, seq, from, to⟩ to toHelp
            else minVersion ← min(minVersion, ver)
    if toHelp ≠ ∅ then
        ver ← F&I(GV)
        for each ⟨t, seq, from, to⟩ ∈ toHelp do
            CAS(psa[t], ⟨?,seq,from,to⟩, ⟨ver,seq,from,to⟩)
            minVersion ← min(minVersion, psa[t].ver)
  ▷4. build
    C_f ← C_n ← new chunk
        with minKey=C.minKey, parent=C, status=infant
    for each chunk C_o from ro.first to last do
        if C_o.minKey ≤ put_key < C_o.next.minKey then
            toPut ← {⟨put_key, GV, put_val⟩}
        else
            toPut ← ∅
        for each k in C_o.ppa ∪ C_o.k ∪ toPut in ascending order do
            if C_n is more than half full then
                C_n.next ← new chunk
                    with minKey=k, parent=C, status=infant
                C_n ← C_n.next
            for each version ⟨ ver, val ⟩ of k, in descending order do
                if val = ⊥ then break                     ▷eliminate tombstones
                insert ⟨ k, ver, val ⟩ to C_n
                if ver < minVersion then break
  ▷5. replace
    do
        C_n.next ← last.next
    while ¬ CAS(last.next+mark, C_n.next+false, C_n.next+true)
    do
        pred ← C's predecessor
        if CAS(pred.next+mark, C+false, C_f+false) then   ▷success
            normalize(C)
            return true
        if pred.next.parent = C then                      ▷someone else succeeded
            normalize(C)
            return false
        rebalance(pred, ⊥, ⊥)                ▷insertion failed, help predecessor
    while true                                            ▷ and retry

procedure NORMALIZE(C)
  ▷6. update index
    for each chunk c from C.ro.first to last do
        index.deleteConditional(c.minKey, c)
    for each chunk c from C_f to C_n do
        do
            prev ← index.loadPrev(c.minKey)
            if c.frozen then break
        while ¬ index.putConditional(c.minKey, prev, c)
  ▷7. normalize
    for each chunk c from C_f to C_n do
        CAS(c.status, infant, normal)
```

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 2, line 32) 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.

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: (1) it scans the PSA for entries with ? whose range overlaps the range covered by the engaged chunks; (2) if any are found, it fetches-and-increments GV and reads its new version into gv; (3) for every psa[t]= $\langle ?, n\rangle$ entry found in (1), it attempts to CAS psa[t] to $\langle$gv, $n\rangle$. Note that scan's CAS (Algorithm 2, line 34) might fail in case it is helped, but either way, it uses the version written by some successful CAS (line 35).

4. Creating new chunks and completing the put. The next stage creates new chunks to replace the engaged ones. 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 C' is created and C_n.next is set to C'. (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: (1) *mark* the next pointer in last as immutable; (2) set the tail of the new chunk sector to its value; and (3) use CAS to set the next pointer of the predecessor of ro.first.

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 (step (1) 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), step (3) CASes the next pointer of the predecessor of ro.first to the next pointer of last.

6. Index update. 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 unindexed, 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

We now provide the key arguments for KiWi's correctness. Due to space considerations, we state the main lemmas without proof, while a formal correctness proof is deferred to the full version of the paper. We begin in Section 4.1 by defining the model and correctness notion we seek to prove, and then present the key safety arguments in Section 4.2. The liveness proof is omitted for lack of space.

4.1 Model and Correctness Specification

We consider an asynchronous shared memory model [36], where a finite number of threads execute memory *operations* on shared objects. 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) operations. High-level operations are *invoked*, then perform a sequence of *steps* on low-level objects, and finally *return*.

Our correctness notion is *linearizability*, which intuitively means that the object "appears to be" executing sequentially. It is defined for a *history*, which is a sequence of operation invoke and return steps, possibly by multiple threads. A history partially orders operations: operation $op1$ *precedes* operation $op2$ in a history if $op1$'s return precedes $op2$'s invoke; two operations that do not precede each other are *concurrent*. In a *sequential* history, there are no concurrent operations. An object is specified using a *sequential specification*, which is the set of its allowed sequential histories. Roughly speaking, a history σ is *linearizable* [25] if it has a sequential permutation that preserves σ's precedence relation and satisfies the object's sequential specification.

KiWi implements a map offering put, get, and scan operations, and 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

Proving KiWi is linearizable is accomplished by identifying, for every operation in a given history, a *linearization point* between its invoke and return steps, so that the operation "appears to" occur atomically at this point. The linearization point of operation op is denoted LP(op).

Puts. We saw above that puts in a chunk C are ordered (lexicographically) according to their $\langle v, j\rangle$ pairs, where $\langle v, i\rangle$ is published in their PPA 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. First, i is published in ppa[t].idx (line 11) and then the location-version pair obtains its final value by a successful CAS of ppa[t].ver, either by the put (line 13) or by a helping thread (line 46). 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 *assigned* $\langle v, j\rangle$ for its key in C.

We note that each put assigns a full version at most once. 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 reading v from GV before t.
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 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 to the same point, they are linearized in increasing full version order.

While a chunk is accessible from the chunks list 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. C is *mutable* if put operations can assign full version in C, otherwise it is *immutable*. An invariant of the rebalance process is that a chunk is immutable before it is accessible from the chunks list and after the freezing stage is completed. In addition, at any point in time each key is covered by the range of at most one mutable chunk. It is easy to show that a put operation always lands at a mutable chunk with a range that covers the key. Thus, 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 4.1. *Consider chunk C accessible as of time t_0, 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 returns v.*
3. *LP(po) is after some operation po′ (possibly po, but not necessarily) publishes for k to 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 for k to C preserve their full version order.*
5. *At time t_0, the value published to k by the put with the latest linearization point before t_0 is associated with the highest full version in C's linked list.*

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, there are three options:

1. If C is not accessible from the chunks list when go starts traversing C's PPA, 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 when go starts traversing C's PPA.
3. Else, let po be the put operation that inserts the value returned by go. LP(go) is the latest between when go starts traversing C's PPA 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). The proof relies on Lemma 5.2 and the rebalance invariants.

LEMMA 4.2. *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.*

Scans are linearized when GV is increased beyond their read point, typically by their own F&I, and sometimes by a helping rebalance. Lemma 5.2 helps to prove the following:

LEMMA 4.3. *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).*

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 5.2 implies the same for puts. It is easy to show that gets and scans land in chunks that contain the saught keys in their ranges. Combined with the rebalancing invariants, Lemma 5.3 shows that get operations satisfy their sequential specification, and Lemma 5.4 proves that scans satisfy their sequential specification. Hence we conclude that KiWi implements a linearziable map.

5. Correctness

5.1 Model

We consider an asynchronous shared memory model [36], where a finite number of threads interact via shared objects. Every thread executes a sequence of operations. An operation's execution consists of a sequence of primitive *steps*, beginning with an *invoke* step, followed by atomic accesses to shared objects, and ending with a *return* step. Steps also modify the executing thread's local variables. We allow read and write steps, as well as atomic read-modify-write steps, such as *compare-and-swap* (CAS), *fetch-and-increment* (F&I) and *fetch-and-add* (F&A).

A *configuration* is an assignment of values to all shared and local variables. A step takes the system from one configuration to another. In the *initial configuration*, each variable holds its initial value.

An *execution* is an alternating sequence of configurations and steps, $C_0, s_1, \ldots, s_i, C_i, \ldots$, C_0 is an initial configuration, and each configuration C_i is the result of executing step s_i on configuration C_{i-1}.

An operation *op* is *pending* in configuration C in a given execution, if the thread executing *op* has yet taken the last step of *op* in the execution leading to C. The *interval of an operation op* is the execution interval that starts at the first step of *op* and ends at the last step of *op*, if there is one, taken by the thread executing *op*. If *op* is pending, then the interval of *op* is the (possibly infinite) execution interval starting at the first step of *op*. Two operations *overlap* if their intervals overlap.

An execution is *serial* if no operations overlap; this means that every operation is executed to completion before another operation starts. Two executions are *equivalent* if every thread in these executions issues the same operations in the same order and gets the same result for each operation.

5.2 Safety

Proving KiWi is *linearizable* [25] is accomplished by identifying, for every operation, a linearization point inside its interval, so that the operation appears to occur atomically at this point. We show that get and scans return the same values as in an equivalent serial execution defined by this linearization.

While a put operation is pending, its entry in the ppa contains the version of the operation, v, and the location of its value, j. Paired, $\langle v, j\rangle$ is called the *location-based version* of the operation. It is also stored in the cell the put operation inserts or updates in the cell linked list. We linearize put operations by the lexicographical order of their location-based version; namely, $\langle v', j'\rangle < \langle v, j\rangle$ if $v' < v$ or $v' = v$ and $j' < j$. Scan operations are linearized when the global version counter is increased beyond their read point. The most subtle linearization is of get operations. We need to make sure it is linearized before all the concurrent puts the operation missed while seeking the value in the chunk. Next we formally define the linearization points of each operation; the linearization point of operation *op* is denoted LP(*op*).

Rebalance operations divide linearization points into epochs: roughly, an epoch starts when a chunk changes its status from infant to normal and ends when the chunk freezes (freezing stage is completed). This is similar to the *freezing point* used to define the linearization points in [12]. Therefore, we describe linearization points of put and get operations in the context of a chunk. Scans are linearized in a chunk-free context. We show a scan observes a consistent view, even when traversing multiple chunks.

Consider chunk c, an operation *op updates* c, if there is a step in which a version v is written into *op*'s entry in c's ppa. Let OP_k^c be the set of put operations writing to k which update c. A put operation $op \in OP_k^c$ first publishes in the ppa the index of its key which holds the value index, j. Then in step σ, v is written (with a successful CAS) in *op*'s entry so it holds *op*'s location-based version $\langle v, j\rangle$.

The linearization point of *op* is derived from the state in configuration C following σ. If in C each cell with key k in the cell linked list and each entry with key k in the ppa have location-based version that is lower than $\langle v, j\rangle$, then LP(*op*) is defined to be the last read step of the global version counter that returns v before σ. Otherwise, let *op*$'$ be the put operation writing to k which published $\langle v', j'\rangle$ in the ppa before σ s.t. $\langle v', j'\rangle$ is the minimal location-based version which exceeds $\langle v, j\rangle$. LP(*op*) is the same as LP(*op*$'$); put operations with the same linearization point are serialized by their location-based version order.

For lack of space, the proofs of the following lemmas are deferred to the full paper.

LEMMA 5.1. *Consider chunk c spanning keys $[k_l, k_u)$, which changes its state from INFANT to NORMAL in step δ_1 and is engaged in a rebalance that completes the freezing stage in step δ_2.*

1. *All cells with key in $[k_l, k_u)$ that where last added to and not removed from the data structure before δ_1 are in c's linked-list in δ_1,*
2. *No put operation can publish its version in c's ppa after δ_2.*

We rely on Lemma 5.1 when proving the next Lemma.

LEMMA 5.2. *Consider chunk c, key k and operation op in OP_k^c, which in step σ publishes $\langle v, j\rangle$ in c's ppa. The following claims hold:*

1. *LP(op) is after op allocates the location j for its value and before σ,*
2. *LP(op) is in a read step of the global version counter that returns v,*
3. *LP(op) is after some operation in OP_k^c (possibly op) with location-based version equal or greater than $\langle v, j\rangle$ is published in c's ppa,*
4. *the linearization points of all operations in OP_k^c preserve their location-based version order.*

Consider a get operation *op* reading key k that starts traversing c's ppa in step τ. If c is not accessible from the chunks list in the configuration preceding τ, then LP(*op*) is in the last step in which c is accessible from the chunks list. Otherwise, c is accessible from the chunks list in the configuration preceding τ. If *op* did not find the key in c then LP(*op*) is τ. Otherwise, let op_l be the put operation which inserted the value returned by *op*. LP(*op*) is the latest between τ and immediately after LP(op_l). It can be inferred from the code and the way linearization points of put operations are defined that LP(op) is in the interval starting in τ and ending when op reads the returned value.

The next lemma proves that no other put operation writing to k is linearized after LP(op_l) and before LP(op). We rely on Conditions 1, 3 and 4 of Lemma 5.2 to prove the next Lemma.

LEMMA 5.3. *Consider a get operation op retrieving the value of key k from chunk c. Let τ be the step in which op starts traversing the* ppa. *The following claims hold:*

1. *If op did not find the key in c, then for each operation $op_m \in OP_k^c$, LP(op_m) is after τ.*
2. *If op returns the value written by operation op_l, then LP(op) is after LP(op_l), and for each operation $op_m \in OP_k^c \setminus \{op_l\}$, LP($op_m$) is either before LP($op_l$) or after τ.*

Consider a scan operation op. LP(op) is the FAI step which increases the global version counter and returns the version number that is later set (with a successful CAS) in op's entry in the global psa. The F&I is done either by the process executing the scan or a concurrent rebalance operation which helps the scan acquire the version. Either way it can be inferred from the use of *aba* counters, that the F&I is done after the scan entry is published in psa, and before the scan reads the version from psa. We rely on Conditions 2 and 4 of Lemma 5.2 to prove the next Lemma.

LEMMA 5.4. *Consider a scan operation op that acquires version v as its read point. For each key k in the range of the scan, op returns the value of the put operation writing to k that is linearized last before LP(op).*

The definition of the linearization points of scans and get operations imply that these operations are linearized within their execution intervals. Condition 1 of Lemma 5.2 implies that each put operation is linearized within its execution interval. Lemma 5.3 implies that get operations satisfy their sequential specification, and Lemma 5.4 implies that scans satisfy their sequential specification. Hence we conclude:

THEOREM 5.5. *KiWi is linearizable.*

5.3 Progress

In Appendix A.2 we prove that KiWi gets and scans are *wait-free*, namely, in any execution, the operation *itself* completes within a finite number of steps by its invoking thread. The proof simply shows that the number of iterations is finite in the loops in these operations. We further prove that KiWi put operations are *lock-free*, namely, in any execution of the operation *some* operation completes within a finite number of steps by the invoking thread. The proof of this property is more subtle. We show that while a put operation can execute infinite number of rebalance and replace methods, some operation (and in fact many operations) completed allocation and made progress.

---

6. Evaluation

6.1 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. This does not violate lock-freedom since the number of threads is much smaller than the chunk size, hence it is impossible for pending puts to fill it up.

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 skiplist [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 $k = 64$.

6.2 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 3 depicts throughput scalability with the number of worker threads. In get-only scenarios (Figure 3(a)), KiWi outperforms the other algorithms by 1.25x to 2.5x. We explain this by the NUMA- and cache-friendly locality of access in its intra-chunk binary search. Under put-only workloads (Figure 3(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

[2] https://github.com/nbronson/snaptree.

[3] http://www.cs.toronto.edu/~tabrown/kstrq/LockFreeKSTRQ.java.

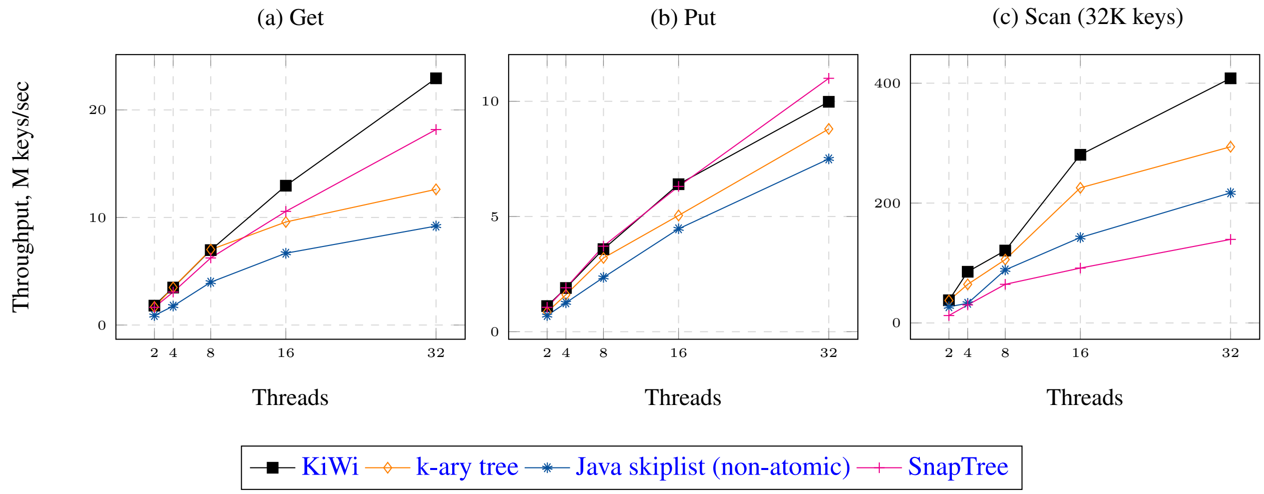

Figure 3: Throughput scalability with uniform workloads, 1M dataset. (a) Get operations, (b) Put operations, (c) Scan operations.

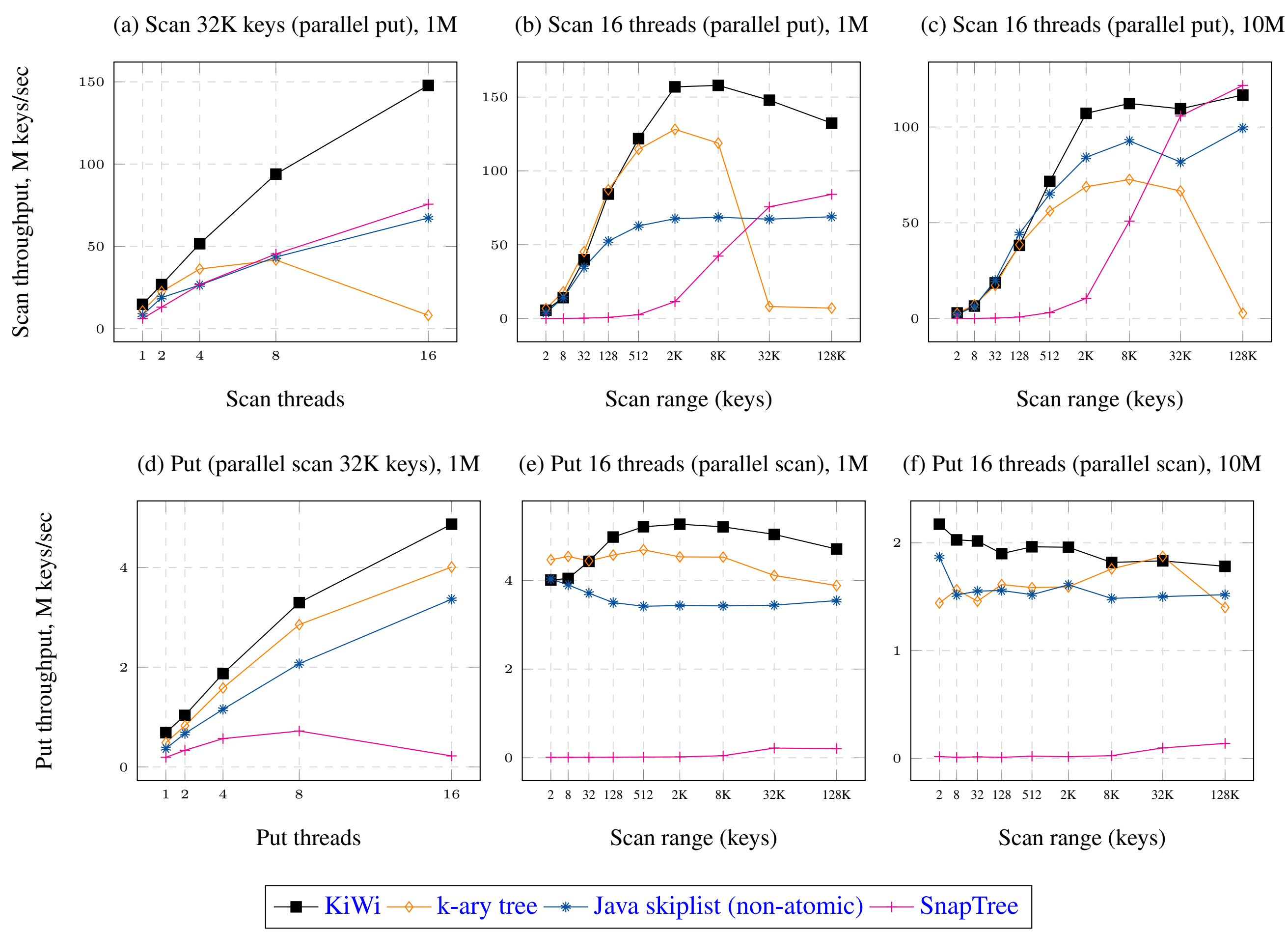

Figure 4: Throughput scalability with concurrent scans and puts. (a,b) Scan operations, 1M dataset. (c) Scan operations, 10M dataset. (d,e) Put operations, 1M dataset. (f) Put operations, 10M dataset.

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 3(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 4(a) depicts scan throughput scalability with the number of threads while scanning ranges of 32K keys. Figure 4(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 4(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 4(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 5 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.

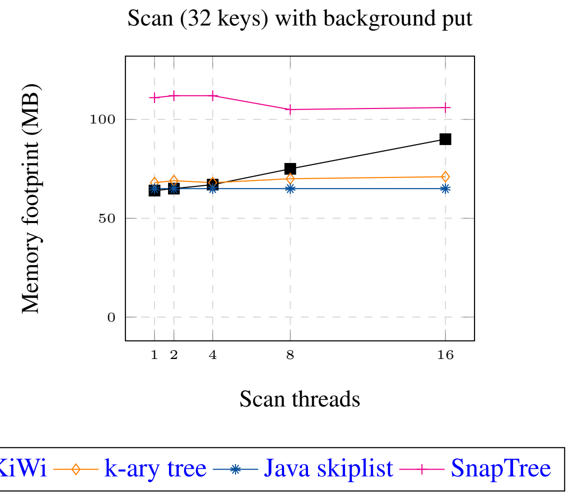

Figure 5: RAM use with parallel scans and puts, 1M dataset.

7. 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.

A. Proof Appendix

A.1 Safety

Proof of Lemma 5.1.

Proof of Lemma 5.2. We consider an execution interval π which spans the execution intervals of all operations in OP_k^c. Denote by $\sigma_1, \sigma_2, \ldots$ the finite sequence of steps of these operations writing versions $v_1, v_2, \ldots$ into entries in the ppa by their order in π, where σ_i is a step of operation op_i; the locations each operation allocated for its value are $j_1, j_2, \ldots$, respectively.

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. Lemma **??** implies that all the cells within the chunk range that were inserted into an earlier chunk were added to the chunk's cell linked list by a rebalance operation that completed before op_1 started. Therefore, these cells have smaller location-based versions and op_1 is linearized in the last read step of the global version counter that returns v_1 before σ_1. Clearly this step is after the put operation is published, and specifically after j_1 is allocated, 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 location-based 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. This step is done after the put is published in the ppa (which is after j_i is allocated) and before σ_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 location-based version order. They are all linearized in read steps of the global version counter returning 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 σ_l before σ_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 location-based version order of the operations, and Condition 4 holds. By Condition 3 of the induction hypothesis, LP(op_l) is after an operation in OP_k^c with location-based version equal or greater than $\langle v_l, j_l \rangle$ is published in c's ppa. 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 $v_l > v_i$. By Condition 2 of the induction hypothesis LP(op_l) is in a read step of the global version counter that occurred after it is set to v_l. op_i eventually obtains the version v_i which is smaller than v_l. This implies op_i published 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 $j_l > j_i$ then op_l allocated j_l after op_i allocated j_i. By Condition 1 of the induction hypothesis, LP(op_l) is after op_l allocated j_l and before σ_l, and LP(op_i) satisfies Conditions 1 and 2.

Proof of Lemma 5.3. It can be inferred from findLatest that *op* returns the value with the maximal location-based version observed by *op* in the ppa and in the cell linked list. In addition, it can be inferred from the code that put operations update values in-place in the cell linked list only if its location-based version is higher than the location-based version of the cell in the list.

First, assume *op* did not find the key in c. By the observations above, all operations in OP_k^c are published in the ppa after τ. Otherwise, *op* should have observed them either in the ppa or in the linked list. By Condition 3 of Lemma 5.2, all operations in OP_k^c are linearized after τ, and Condition 1 holds.

Next, assume *op* returns the value written by operation op_l; the location-based version of op_l is $\langle v_l, j_l \rangle$.

If c is not accessible from the chunks list in the configuration preceding τ, then LP(*op*) is in the last step in which c is accessible from the chunks list. By Claim 2 of Lemma 5.1 no put operation can publish its location-based version in c's ppa after τ, and by Condition 1 of Lemma 5.2 LP(*op*) is after LP(op_l). Otherwise, it is clear by definition that LP(*op*) is after LP(op_l).

Consider an operation $op_m \in OP \setminus \{op_l\}$ with location-based version $\langle v_m, j_m \rangle$. By Condition 4 of Lemma 5.2, if $\langle v_m, j_m \rangle < \langle v_l, j_l \rangle$ then LP(op_m) is before LP(op_l). It is left to show that if $\langle v_m, j_m \rangle > \langle v_l, j_l \rangle$ then LP(op_m) is after τ. By the observations above, all operations $op_m \in OP_k^c \setminus \{op_l\}$ with location-based version $\langle v_m, j_m \rangle > \langle v_l, j_l \rangle$ are published in the ppa after τ, and hence are not observed by *op*. Condition 3 of Lemma 5.2 implies that these operations are linearized after at least one of them is published in the ppa, hence Condition 2 also holds.

Proof of Lemma 5.4. It can be inferred from findLatest that for each key k in the range of the scan, *op* returns the maximal location-based version of k that does not exceed the scan read point observed by *op* in the ppa and in the cell linked list. In addition, it can be inferred from the code that put operations update values in-place in the cell linked list only if its location-based version is higher than the location-based version of the cell in the list.

Consider a put operation op_m that writes to a key in the range of the scan but is not observed by *op*. If op_m acquires version that is less than v then it acquired a version before the scan increased the global version counter. Since *op* did not observe op_m in the ppa, op_m completed before *op* read the entry in the ppa, and since *op* did not observe op_m in the linked list, then the location-based version of the cell with key k already in the linked list is higher than the location-based version of op_m. By Condition 4 of Lemma 5.2, op_m is linearized before the put operation writing the value returned by the scan.

Finally, by Condition 2 of Lemma 5.2 all put operations that are not observed by *op* and acquire version that is greater than v are linearized after LP(*op*).

A.2 Progress

We now show that *get* and *scan* operations are wait-free, and *put* operations are lock-free.

We say that a sequence π_s is a *sub-execution*, if π_s is a suffix (or a prefix) of an execution. We say that thread t is *running* in a sub-execution π_s, if at least one step in π_s is executed by thread t. Given two keys K_1 and K_2, we write $\mid K_2 - K_1 \mid$ to denote the number of possible keys between K_1 and K_2 (notice that $\mid K_2 - K_1 \mid$ is always a finite number because each key is stored in a bounded number of bytes). Given a chunk X, we write $min(X)$ to denote the minimal key in X (as mentioned before, $min(X)$ is never changed during the lifetime of X).

LEMMA A.1. *Let $\pi = C_0, s_1, C_1, \ldots, s_i, C_i, \ldots$ be an execution. Let X and Y be two chunks in configuration C_i such that $Y = X.next$. For any configuration C_j such that $j \geq i$ we have $min(X) < min(Y)$ in C_j.*

Proof The algorithm writes the address of Y in $X.next$ only if $min(X) < min(Y)$ (this happens either at line 22 of the function *Kiwi::Replace*, or within the function *balance()*). Since $min(X)$ and $min(Y)$ are never changed, $min(X) < min(Y)$ in any C_j such that $j \geq i$.

LEMMA A.2. *The function* get *is wait-free.*

Proof We prove that *get* is wait-free by showing that the functions *Kiwi::FindChunk* and *Chunk::Find* are wait-free.

The function *Kiwi::FindChunk* has a single loop: we write $next_i$ to denote the value of variable *next* at the beginning of the i-th iteration of the loop. Because of Lemma A.1, $min(next_i) < min(next_{i+1})$. Hence the loop terminates after at most $\mid key - min(next_1) \mid$ iterations. Since all the functions invoked by *Kiwi::FindChunk* are wait-free, each invocation of *Kiwi::FindChunk* by thread t completes within a finite number of steps by t.

The function *Chunk::Find* invokes the functions *Chunk::FindInPending* and *Chunk::FindInCellList*. The loop in *Chunk::FindInPendingList* completes after a constant number of iterations. The function *Chunk::FindInCellList* goes over the chunk's link-list exactly once (this link-list has at most M cells). Hence, each invocation of *Chunk::Find* by thread t completes within a finite number of steps by t.

LEMMA A.3. *The function* scan *is wait-free.*

Proof All functions invoked by *scan* are wait-free. It is sufficient to show that the loops in *scan* have a finite number of iterations.

The function *scan* has two nested loops: we write L_e to denote the external loop (begins at line 10), and L_i to denote the internal loop (begins at line 12). Let key_i be the value of variable *key* at the end of the i-th iteration of L_e. Each key_i is equal to the minimal key of the chunk which is handled by iteration $i + 1$ (this chunk is pointed by the variable *chunk*). Because of Lemma A.1, if L_e has (at least) $i + 1$ iterations then $key_i < key_{i+1}$. Hence, L_e has at most $\mid maxKey - minKey \mid$ iterations.

The loop L_i goes over the link-list of a chunk (this link-list has at most M elements), hence L_i has at most M iterations.

Therefore, each invocation of *scan* by thread t completes within a finite number of steps by t.

LEMMA A.4. *The function* put *is lock-free.*

Proof We prove by contradiction. Assume that *put* is not lock-free. Hence there exists an infinite execution $\pi = C_0, s_1, C_1, \ldots$ such that after configuration C_{k0} ($k0 \geq 0$) no operation completes and no new operation is invoked. We have already shown that *get* and *scan* are wait-free, therefore there exists configuration C_{k1} in π ($k0 \leq k1$) such that after C_{k1} all the running threads execute *put* operations.

We write $t : X$.*allocate* to denote an invocation of *allocate* on chunk X by thread t (*allocate* is invoked by function *put* at line 6). We say that invocation *t:*X.*allocate* is *successful* if it returns a reference to a valid cell (i.e., it returns a non-null value).

Consider an execution of the loop in function *put* by thread t. If $t : X$.*allocate* is not successful at iteration $i + 1$ ($i \geq 1$) of t , then at iteration i the invocation of $t : Y$.*allocate* is also not successful. Hence thread t invokes *Rebalance(*Y*)* at the end of iteration i. Therefore $Y \neq X$ and chunk X has been added to kiwi (at some point) during iterations i and $i + 1$ of t (when a chunk is added to kiwi, this chunk is not full). Therefore another thread $t' \neq t$ successfully invoked $t' : X$.*allocate* (at some point) during iterations i and $i + 1$ of t (otherwise $t : X$.*allocate* should be successful). Since a thread do not start new iteration of this loop after a successful invocation of *allocate* (see lines 6–9 in *put*), we know that there exists configuration C_{k2} in π ($k1 \leq k2$) such that: after C_{k2} the *put* operations do not start new iterations of this loop.

Since *allocate* is wait-free, there exists configuration C_{k3} in π ($k2 \leq k3$) such that no thread executes *allocate* after C_{k3}. Hence, no chunk becomes full in π after configuration C_{k3}.

In the following paragraphs we focus on the functions *Kiwi::Rebalance* and *Kiwi::Replace*. Notice that *Kiwi::Rebalance* calls to *Kiwi::Replace*; and that *Kiwi::Replace* recursively calls to *Kiwi::Rebalance* in line 20. An invocation of *Kiwi::Rebalance* on chunk X removes X from kiwi: hence for each thread t and chunk X, *Rebalance(*X*)* may be invoked at most once by t.

Let N_{puts} be the number of *put* operations after configuration C_{k3}. Let N_E be the maximal number of new chunks created by an invocation of *balance()* (in our implementation $N_E = 4$). Since after C_{k3} no chunk may become full, at most $N_{puts} \times N_E$ new chunks may be created after C_{k3}. Therefore, after configuration C_{k3} the threads invoke function *Kiwi::Rebalance* a finite number of times. Hence there exists a configuration C_{k4} in π ($k3 \leq k4$) such that *Kiwi::Rebalance* is not invoked after C_{k4}.

Since the other functions invoked by *put* are lock-free[4], there exists a configuration C_{k5} in π ($k4 \leq k5$) such that all the running threads after C_{k5} execute the loop in *Kiwi::Replace* (and this loop is never terminated). Hence, the CAS at line 22 always fails after configuration C_{k5} (i.e., no CAS updates shared memory after C_{k5}). This is a contradiction, because the CAS at line 22 may fail only if shared memory has been updated since the beginning of the iteration.

[4] The other functions invoked by *put*, *Kiwi::Rebalance* and *Kiwi::Replace* are lock-free: we assume that *index.replace* is lock-free, the other ones are trivially wait-free.