We thank the reviewers for the helpful and insightful comments. We have addressed their concerns, as we now explain.

Reviewer #1:

The paper lacked a cover letter explaining the differences with the original PPoP paper.

We apologize for not clarifying the difference. As the referee noticed, the main difference relative to the PPoPP version is the correctness proof. This includes:
- rigorous model definitions in Section 4.1, as opposed to an informal model in PPoPP;
- a formal linearizability proof in Section 4.2, whereas the PPoPP version only informally discusses linearization points and states two lemmas without proving them; and
- a formal liveness proof (Section 4.3), which was omitted from the PPoPP version for lack of space.

Reviewer #2:

The evaluated implementation isn't non-blocking, because the extended index API that KiWi needs is implemented using locks ... *It's important to know if the lazy indexing optimization is compatible with a fully non-blocking implementation, ...*

We added the following clarification to the paper:

> We note that it is straightforward to implement such a non-blocking index using the extended load-linked and store-conditional primitives, LLX and SCX, due to Brown et al. [13]; a general approach for implementing non-blocking search trees using these primitives is given in [14]. In our implementation, we instead use locks. This does not affect performance as index nodes are locked infrequently (only upon rebalance), and only by background threads off the main execution path.

The title, abstract and introduction motivate KiWi with a data analytics use case, but the evaluation doesn't demonstrate suitability for data analytics. … I actually don't think that the analytics motivation is very important, given KiWi's contribution over other concurrent maps. But if the claim is made, it should be backed up.

KiWi was motivated by analytics use-cases and made important strides towards addressing the challenges they face. In particular, KiWi was the first KV-store to provide good performance for real-time analytics-style workloads involving long scans in parallel with real-time updates. While we did not run a full-blown analytics data set, the 10M KV-pair data set in our experiments was bigger than ones used in earlier works (note that this is a journal version of a PPoPP 2017 paper, whose submission deadline was in August 2016). KiWi's chunk-based organization was the basis for follow-up work on Oak, to appear in PPoPP 2019, which pushes the envelope further by taking data off heap to support full-blown analytics workloads.

In the revision, we have toned down and focused the claims about analytics, as follows:

- We removed the first paragraph of the abstract and no longer mention analytics there.
- We removed the analytics application examples in the 2nd paragraph of the introduction.
- We refer to systems requiring long scans in parallel with updates as “real-time analytics”. This is now the only place where the word “analytics” appears in the introduction.
- We did not change the title since we wish to keep it the same as in the conference version.

Compare to "Harnessing epoch-based reclamation for efficient range queries".

We added the citation (in the related work section). Please note that it appeared a year after the conference version of our paper.

Page 5, line 36: It's not clear why pending operations are relevant at this point.

We have added the following explanation:

> To ensure progress, we need to make sure that before rebalance happens, at least one thread successfully completes a put to the chunk, even if threads invoke operations that take up slots in the arrays and remain pending without ever completing. To this end, we set the number of entries in the k and v arrays to be larger than twice the number of threads.

This is used in the liveness proof (towards the end of Section 4.3).

Reviewer #3:

1. Related papers.

The referee mentioned three related papers, all of which were published after the conference version of this work. We now discuss these papers in our related work section (citations [8,20,39]).

2. The primitives (functions) "put", "get", "scan", etc., are not using a different font in the paper. They look confusing in some contexts because they usually are verbs. I would suggest using a different font, or call then "put operation", "get operation" and "scan operation".

We prefer to use gets, puts, and scans as English nouns, hence write them without a special font. We use the code font when we refer to the API call (with the parameters). We feel that this style has good readability, though we do recognize that the choice is subjective.

3. How does the implementation collect old versions during execution?

Old versions are removed during rebalance, as explained under the heading “4. Creating new chunks and completing the put” in Section 3.3.2.

4 Evaluation

4.1 Key distribution

We have added the following discussion to the methodology paragraph in Section 5.1

> For the most part, we use the same methodology as previous works, which is the methodology supported by synchrobench, where keys are selected uniformly at random. We additionally study an ordered workload to illustrate the benefit of using a balanced data structure as opposed to an unbalanced one.
> We also experimented with a skewed distribution, where after populating the data structure with a uniform distribution over the full key range in the warmup phase, the measured experiments exercise only 10% of the key range. The results were very similar (in absolute number as well as in trends) to the uniform distribution. For example, in one set of experiments with 4 update threads and 4 threads running scans of different ranges (10 to 25K keys), the put throughput in the skewed key distribution was within 2% of the put throughput in the uniform one across all scan sizes. Scan times exhibited higher variability -- up to 15% -- but did not show a clear advantage in one key distribution over the other. Rather, in some experiments the scan was faster over the skewed distribution and vice versa.

Note that synchrobench does not support a Zipf distribution, so we generated the skew by uniformly accessing a “hot” sub-range of the keys.

4.2 Since there is no worst-case time bound for KiWi, it is worth studying the performance of several bad cases to understand how much these would affect performance. For example, 1) What happens when the workload or the setting invokes more frequent vs. less frequent rebalances?

The rebalance frequency is controlled by KiWi parameters and is not sensitive to the workload.

We have added the following two discussions to the implementation paragraph in Section 5.1, regarding chunk size:

> We have found this chunk size to be big enough to offer good locality. Bigger chunks favor puts albeit with diminishing returns, and mildly slow down gets and scans because larger chunks allow more ``imbalance'' (obsolete data to iterate through, long bypasses in the search) than smaller ones.

And regarding rebalance parameters:

> The probability for invoking rebalance and the threshold on the ratio of the batched prefix control the aggressiveness of rebalancing. On the one hand, aggressive rebalancing reduces put throughput because put threads shoulder the burden of rebalancing. On the other hand, it favors gets and scans because it shortens search times and reduces iteration through deleted or over-written items. We have chosen these parameters so as to strike a balance between the two.

2) What about when searches often follow a long chain in a chunk (e.g., almost linear time to the chunk size). In other words, how much does the batched prefix improve the performance?

We have added the following to the discussion of the batched prefix in Section 3.1:

> Empirically, we have found that the batched prefix improves performance by an order of magnitude.

3) How is the performance affected by how much the scans and updates interleave with each other?

This can be observed from the difference between Figure 4 (where scans and puts do not interleave) and Figure 5 (where they do). For example, in the 1M dataset, the 16-thread put throughput without background scans is approximately 6.4M ops/sec (Figure 4b), whereas with background scans of different lengths it varies between 4M and 5M ops/sec (Figure 5e).

The range length somehow reflects this, but would also change the number of I/Os so the performance is not purely affected by the scan-update interleaving.

I/Os are very infrequent because KiWi is an in-memory data structure.

4.3 The parameters are specified at the beginning of Section 5. Are they selected based on the machine, or just empirically, or because of other reasons? What will happen if a different number is chosen? Is the performance sensitive to these parameters?

Please see our answer to 4.2 above.

Minor comments:
** 3.2.1: "Scan then uses the fetched version, ver": Is this ver the value read from GV? It is not defined above.*
Fixed (now defined in the preceding sentence).

** 3.2.2: the term "frozen" is used but not previously defined or descriptions.*
Added explanation about rebalance-related data in the chunk at the end of Section 3.1.

Typos:
** P1: KV-maps have become centerpiece -> KV-maps have become a centerpiece*
** P2: KiWi is a balanced data strucutre -> KiWi is a balanced data structure*
** P4: when scans are onging -> when scans are ongoing*
** P7: Scan then uses the fetched version -> The scan then uses the fetched version*
All fixed.