Brief Announcement: A Scalable Recoverable Skip List for Persistent Memory

Sakib Chowdhury
University of Waterloo
Waterloo, Ontario, Canada
sakib.chowdhury@uwaterloo.ca

Wojciech Golab
University of Waterloo
Waterloo, Ontario, Canada
wgolab@uwaterloo.ca

ABSTRACT

Interest in recoverable, persistent-memory-resident (PMEM-resident) data structures is growing as availability of Intel Optane Data Center Persistent Memory increases. An interesting use case for in-memory, recoverable data structures is for database indexes, which need high availability and reliability. RECIPE, a popular conversion technique to make existing, proven-correct algorithms recoverable, is limited to certain classes of algorithms and does not prescribe how to reference data stored in relocatable regions of memory.

The Untitled Persistent Skip List (UPSkipList) is a PMEM-resident recoverable skip list derived from Herlihy et al.'s lock-free skip list algorithm. It is developed using a new conversion technique that extends the RECIPE algorithm by Lee et al. to work on lock-free algorithms with non-blocking writes and no inherent recovery mechanism. The algorithm is also extended to support concurrent data node splitting to improve performance.

Comparison was done against the BzTree of Arulraj et al., as implemented by Lersch et al., which has non-blocking, non-repairing writes implemented using the persistent multi-word CAS (`PMwCAS`) primitive by Wang et al. Tested with the Yahoo Cloud Serving Benchmark (YCSB), UPSkipList achieves better performance in write-heavy workloads at high levels of concurrency than BzTree, showing that the extension to RECIPE is an effective alternative.

CCS CONCEPTS

• **Information systems → Data structures**; **Key-value stores**; **Storage class memory**; **Indexed file organization**; • **Hardware → Non-volatile memory**.

KEYWORDS

concurrency; persistent memory; skip lists; data structures; scalability

ACM Reference Format:
Sakib Chowdhury and Wojciech Golab. 2021. Brief Announcement: A Scalable Recoverable Skip List for Persistent Memory. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3409964.3461819

1 INTRODUCTION

Persistent memory (PMEM) is an emerging technology that combines the best aspects of volatile main memory with secondary storage by supporting byte-addressable access to non-volatile primary storage at low latencies, allowing the CPU to reasonably stall while waiting for data [16]. One notable application for PMEM is in database systems, where successful recovery from crash failures is critical in maintaining the integrity of the database [9]. Theoretically, PMEM can reduce the need for backups and logging to stable storage by making modifications to data structures persist across power failures as soon as they are flushed out of CPU caches and store buffers.

Building lock-free, recoverable index structures from scratch is difficult, so they are generally developed using either a transformational or transactional approach. Transformational techniques like RECIPE [11] have been developed that prescribe methods of converting several classes of existing lock-free algorithms to be recoverable. However, this does not work for algorithms with non-blocking writes that do not fix inconsistencies by helping finish operations, which is a category that encompasses many algorithms that would be useful to convert, including Herlihy et al.'s lock-free skip list algorithm [10]. Transactional approaches like the use of persistent multi-word CAS (`PMwCAS`) [17] are simpler to implement and support lock-free non-repairing algorithms. As shown in the evaluation by Lersch et al., however, `PMwCAS` has scalability limits for write-heavy workloads [12]. This motivates the development of a method to transform algorithms with non-blocking, non-repairing writes without introducing performance bottlenecks.

We have developed a recoverable skip list that scales across multiple NUMA nodes. It is an adaptation of Herlihy et al.'s lock-free skip list, which has non-blocking writes that do not fix inconsistencies, and could not have been implemented using only RECIPE. We have also modified the algorithm to improve performance by storing multiple keys in a single node, achieving better cache efficiency.

The contributions of this paper are as follows:

1. Untitled Persistent Skip List (UPSkipList), a fully-PMEM-resident skip list, implemented using an extension to RECIPE that allows the conversion of lock-free algorithms with non-blocking writes that do not fix inconsistencies.
2. Experimental results showing that this conversion method has benefits over using `PMwCAS` to implement algorithms with non-blocking, non-repairing writes.

2 ALGORITHM

The UPSkipList algorithm uses as a starting point the lock-free skip list design by Herlihy et al. [10], which satisfies the time complexity and data structure requirements of Pugh's skip list [15]. We modified this algorithm not only to achieve recoverability, but also to store multiple keys in a single node through the implementation of recoverable concurrent node splits.

RECIPE requires that writes in a lock-free algorithm satisfy at least one of three properties [11], and prescribes a conversion method for the operations satisfying that property. In a skip list, the operations that need to be converted are updates and inserts.

The first property applies modifications using a single atomic store to perform an update. Though this situation is trivial to make recoverable, and applies to updates, insertions in Herlihy et al.'s lock-free skip list do not involve just a single atomic store, and instead require the use of multiple atomic steps to perform the operation.

The second and third properties require that writes are either non-blocking and repairing, or blocking and non-repairing. Neither applies to Herlihy et al.'s skip list algorithm. Writes do not fix inconsistencies of insertions when they are found; they simply assume that another thread is taking care of it. Reads only fix inconsistencies as part of the algorithm when removing links to logically-deleted nodes. Writes are non-blocking as well, which prevents detection of inconsistent nodes in need of repair by checking whether the nodes are locked, leaving RECIPE's third conversion method unusable for this algorithm.

2.1 Extending RECIPE to support non-blocking, non-repairing writes

To apply RECIPE's third conversion technique, it is necessary to find a way to detect when an inconsistency is due to a crash failure without having to check a DRAM-resident lock, as required by RECIPE.

Although volatile locks can be added to Herlihy et al.'s algorithm for this purpose, this would require that during recovery these locks be reinitialized and reassociated with the nodes, making recovery time dependent on the size of the structure. Instead, failure detection can be done using a method similar to that proposed by Aguilera et al., using an epoch number associated with an object whose consistency is being verified [1]. This method of failure detection is implemented by Golab and Ramaraju to construct recoverable mutexes [8], and a form of it can be seen implemented in BzTree [2] just for the detection of interrupted node resizes.

Each failure-free period is tracked using a monotonically increasing PMEM-resident variable called the `epochID`. During creation or confirmation of consistency of a node, the `epochID` of the node is set to match the current epoch. After a failure, the `epochID` of a node will be out of date. Any thread that comes across it will know that there are no other threads responsible for fixing any inconsistencies with that node, which would be the case if the `epochID` did match the current epoch. This replaces the method used by RECIPE to detect if a node's inconsistencies will be repaired, which they do by checking for a held lock [11]. When traversing the structure, an outdated `epochID` is first updated by a thread trying to claim it so that it can restore consistency, preventing multiple threads from trying to repair the same node.

An example procedure for recovering nodes is shown in Function 1. This function is called on every node read by a thread during traversal. If the node being examined is outdated, the thread will try to claim the right to recover it by updating its `epochID`. If successful, this thread will then check for all possible inconsistencies. With the ability to discern incomplete non-blocking writes from failed ones using only PMEM-resident variables, it is now possible to follow the rest of the third RECIPE conversion method to achieve recoverability by implementing recovery methods for these inconsistencies.

Function 1: CheckForRecovery(Node currentNode)

```
1 nodeEpoch = currentNode.epochID;
2 if nodeEpoch does not match root.epochID then
3     if not currentNode.epochID.CAS
        (nodeEpoch, root.epochID) then return;
4     RepairInconsistencies (currentNode);
5 end
```

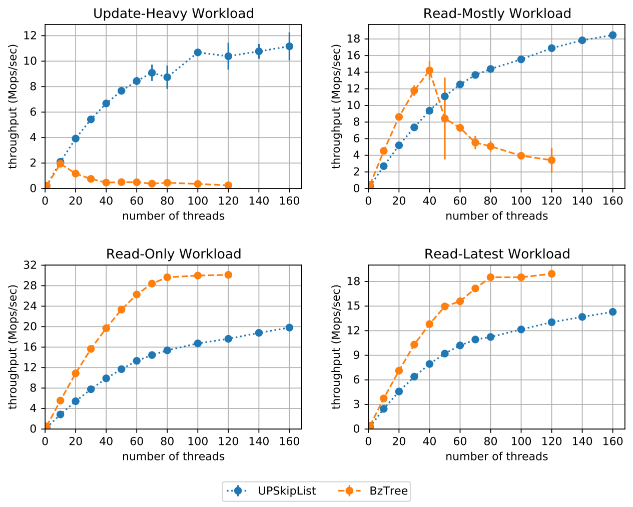

Figure 1: Throughput comparison using YCSB benchmark of UPSkipList and BzTree. Each point is the average of three runs, and error bars indicate one standard deviation.

3 EXPERIMENTAL RESULTS

To evaluate the performance of the algorithm after conversion, we tested throughput comparing UPSkipList with BzTree [2]. BzTree, built using the Wang et al.'s persistent multi-word CAS (`PMwCAS`) recoverable primitive [17], also has non-blocking writes, with the usage of `PMwCAS` removing the need to repair inconsistencies by keeping them hidden from the program. The implementation of BzTree used is by Lersch et al., who benchmarked index data structures on real persistent memory, and found BzTree to perform the best of the PMEM-only data structures they tested [12].

Testing was done on an 80-core, 160-thread, 4-socket Intel Xeon machine with Intel Optane DC Persistent Memory. BzTree, which is not NUMA aware, was run using a single memory pool striped across Optane devices on multiple nodes. UPSkipList was also run in this configuration, although it supports separate pools as well.

Due to internal limitations of the PMwCAS library used by Lersch et al., this implementation of BzTree does not support being run with more than 120 threads. Threads were assigned to NUMA nodes in a round-robin manner, ensuring an equal distribution across all nodes for all numbers of threads. All physical cores were filled first, with remaining threads being assigned to hyperthread siblings. The Yahoo Cloud Serving Benchmark (YCSB) was used to generate workloads for the runtime tests, which are all Zipfian except for the read-latest workload [5]. The data structures were preloaded with 100 million key-value pairs prior to each test.

The results in Figure 1 show that our conversion technique results in a recoverable data structure that is competitive with existing work. BzTree outperforms UPSkipList at both read-only and read-latest workloads. The reason for this is having a more efficient lookup process inside nodes. In BzTree, after a node split, both nodes contain sorted keys, while keys inserted between splits are stored unsorted in an overflow region. BzTree's lookup process takes advantage of this, using a binary search within the n sorted keys to attempt to find a key in $O(\log(n))$ time, and then linearly searching in the overflow region if necessary, while UPSkipList maintains keys in an unsorted manner, always requiring a linear search. The sorting optimization can be implemented in UPSkipList, due to similarity in the implementation of node splits, and will be done in future work.

Comparatively, BzTree does poorly at higher update-to-read ratios, which is a result matching that of Lersch et al. [12]. Similar to their observations, BzTree scales up to a point, after which performance falls off. The level of concurrency BzTree scales up to is inversely correlated with the proportion of updates. The reason BzTree's performance falls off is due to its internal use of PMwCAS to perform its writes. Where UPSkipList manages to update a key using a single CAS operation, a BzTree thread needs to use PMwCAS to change the key value to ensure it does not interfere with a concurrent PMwCAS operation and can perform the update safely. This requires interaction with data structures internal to PMwCAS, which does not impact performance until reaching a certain level of contention. This is a tradeoff of using PMwCAS to perform non-blocking writes, which adds a potential bottleneck for some patterns of usage in exchange for simpler algorithms and correctness. Adapting an existing, mature, non-recoverable algorithm using RECIPE also provides a simpler implementation and correctness reasoning, and our extension to detect inconsistencies in algorithms with non-blocking, non-repairing writes reduces the need to use external libraries that may add bottlenecks like PMwCAS.

4 RELATED WORK

The skip list [15] for this paper is based on the lock-free skip list by Herlihy et al. [10]. Multiple recoverable skip lists have been built [4, 7, 17]. Some use hybrid designs [4, 7], removing the possibility of instantaneous recovery. Wang et al. use their software-based persistent multi-word CAS (PMwCAS) primitive [17], and in comparison to the B-tree they built with the primitive, their skip list achieves only 20-40% of the performance.

More generally, multiple recoverable index structures have been developed [2, 14, 17, 18], some of which have been evaluated on real PMEM hardware by Lersch et al. [12]. Of the ones tested, BzTree [2] appears to be the most comparable to UPSkipList, due to being the fastest PMEM-only structure they tested. It builds on the BwTree algorithm [13] made recoverable using PMwCAS [17].

Alternative conversion techniques to RECIPE [11] include the non-transactional log-free primitives of David et al. [6] whose use is more involved than RECIPE, and NVTraverse [7], where a skip list can only be made recoverable in a hybrid manner using both DRAM and PMEM, increasing recovery time. A hybrid indexing technique using concurrent skip lists has also been explored by Balmau et al. with FloDB [3].

ACKNOWLEDGMENTS

This research is supported by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 2000. Failure Detection and Consensus in the Crash-Recovery Model. *Distrib. Comput.* 13, 2 (April 2000), 99–125.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565.

[3] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proc. of the Twelfth European Conference on Computer Systems (EuroSys '17).* 80–94.

[4] Qichen Chen, Hyojeong Lee, Yoonhee Kim, Heon Young Yeom, and Yongseok Son. 2019. Design and implementation of skiplist-based key-value store on non-volatile memory. *Cluster Computing* 22, 2 (01 Jun 2019), 361–371.

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing (SoCC '10).* 143–154.

[6] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proc. of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18).* 373–385.

[7] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proc. of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20).* 377–392.

[8] Wojciech Golab and Aditya Ramaraju. 2016. Recoverable Mutual Exclusion [Extended Abstract]. In *Proc. of the 2016 ACM Symposium on Principles of Distributed Computing (PODC '16).* 65–74.

[9] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317.

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

[11] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proc. of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).* 462–477.

[12] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 574–587.

[13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proc. of the 2013 IEEE International Conference on Data Engineering (ICDE 2013).* 302–313.

[14] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proc. of the 2016 International Conference on Management of Data (SIGMOD '16).* 371–386.

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

[16] Andy Rudoff. 2017. Persistent Memory Programming. *USENIX ;login:* 42, 2 (2017), 34–40.

[17] T. Wang, J. Levandoski, and P. Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proc. of the 2018 IEEE 34th International Conference on Data Engineering (ICDE).* 461–472.

[18] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. 2016. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Trans. Comput.* 65, 7 (2016), 2169–2183.