JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 13, No. 4, December 2019, pp. 141-150

A Low Latency Non-blocking Skip List with Retrial-Free Synchronization

Eunji Lee*
Department of Smart Systems Software, Soongsil University, Seoul, Korea
ejlee@ssu.ac.kr

Abstract

The underlying technology trend stresses the design of the software. With increasing use of many-core computers that equip a large number of independent processor units, enhancing scalability and concurrency of commercial software is of crucial importance. To fulfill this demand, non-blocking implementations of the popular data structures are extensively explored both in academia and industry to effectively harness the massive parallelism in a many-core system. This paper presents a new non-blocking skip list that is not only scalable but also provides low latency even under high-concurrency pressure. Existing techniques for parallelizing skip lists rely on retrying insertion operations which fail because one thread interferes with another. This approach can introduce long-tail latency when multiple threads compete for access to the same links. We address this issue by exploiting the probabilistic nature of the skip list by allowing insertion operations to terminate after a failure, even if all the links from a node have not been updated. The resulting reduction in the heights of many nodes changes the statistical properties of the links, on which the efficiency of the skip list depends. We compensate this side-effect by recording reductions in node height and recompense for them when new nodes are created. To demonstrate the effectiveness of our approach, we implement a prototype of our low-latency non-blocking skip list, and the measurement study with various workloads shows that our skip list provides more scalable performance and lower tail latency compared to existing skip lists.

Category: Databases / Data Mining

Keywords: Skip list; Data structures; Database systems; Concurrent computing; Scalability; NoSQL systems

I. INTRODUCTION

Computer hardware technologies have undergone massive improvements over the last decades. These advancements have been fueled by the exponential improvement in the transistor density, which is generally referred to as Moore's law [1]. More recently, however, the increase in performance comes instead from on-die parallelism. Since the power of individual processors is limited by the scalability of silicon processes and power consumption, efforts to develop faster computers are increasingly focused on multi-core processors [2-4].

However, parallelization of algorithms and synchronization of parallel processes are well known complex problems. Parallel programming requires an intricate analysis of the underlying algorithms and a complicated synchronization protocol is needed to mediate concurrent execution flows. While numerous studies have been carried out to improve the scalability of the computing systems [5-21], the parallel programming still remains a challenge.

In line with previously reported studies, our work is

Open Access http://dx.doi.org/10.5626/JCSE.2019.13.4.141 http://jcse.kiise.org

Received 09 December 2019; **Accepted** 17 December 2019
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

Table 1. Data structures in NoSQL systems

Data store	In-memory structure	On-storage layout
LevelDB	Skip list	LSM tree
RocksDB	Skip list	LSM tree
HBase	Skip list	LSM tree
DynamoDB	Skip list	LSM tree
Memcached	Hash table	Archival log
Redis	Hash table	Archival log

motivated by an observation that current skip lists have limited scalability despite continual improvements over the years. The skip list is a multi-layer linked list which offers a good compromise between complexity and performance; insertion and lookup both have logarithmic time complexity. It is extensively used in key-value and NoSQL databases [22-27], as summarized in Table 1, while its adoption is also being considered in Linux Kernel, for replacing a few previous data structures in task scheduling component and the file system of recent Linux kernels [28-30].

However, the type of skip list currently used in commodity software lacks scalability. Inserting data into a skip list involves multiple pointer updates, which are difficult to perform atomically without using a mutex lock. Thus, LevelDB [22], a key-value store (KVS) developed by Google, uses a blocking version of a skip list that relies on the condition variable to provide mutual exclusion. This unscalable implementation of the skip list severely limits the parallelism of applications, thereby failing to support scalability at high concurrency. More recent KVSs such as Facebook's RocksDB have a more scalable memory component with a non-blocking skip list [9, 10], which allows concurrent writes without a mutex lock. However, this skip list runs in an infinite loop until all the updates in a transaction are atomically completed. Under high concurrency, inopportune interleaving of threads causes failures and undesirable increases in latency.

To address these issues, we present a low-latency non-blocking skip list, which exploits the probabilistic nature of the skip list to eliminate long and unpredictable delays under concurrent writes. The heights of the nodes in a skip list (i.e., the number of pointers pointing to other nodes) are randomly determined using a probability function so that they have a set distribution. Changing this distribution does not affect the correctness of a skip list as it simply produces a marginal reduction in performance. We exploit this property to reduce the latency of concurrent transactions. During a transaction, we update pointers from the bottom up, and if a pointer update fails due to interference from other threads, it is never retried. Instead, the height of the node is simply reduced to the number of pointers we have successfully updated. This mechanism prevents continual failures by multiple threads causing large increases in latency, even under high concurrency.

However, repeated changes in the node heights resulting from a carefully built probabilistic function will increase the time complexity of the skip list. We remedy this side-effect by run-time tuning of the probability with which different node heights are selected: the probability associated with the height that a node would have had before the intervention of our mechanism is increased and that associated with the new height produced by our mechanism is reduced. Thus, the intended distribution is restored and maintained.

To assess the effectiveness of our approach, we implement three types of skip lists, including the proposed skip list, which we evaluate under various workloads. The experimental results show that our skip list provides more scalable performance with lower tail latency than current versions of a skip list under a highly concurrent workload.

The remainder of this paper is organized as follows. In Section II, we provide a brief overview of the skip list. In Section III, we explain the design and implementation of our low-latency non-blocking skip list. In Section IV, we describe the evaluation methodology and present experimental results. In Section V, we discuss the proposed design in relation to prior work, and in Section VI we draw conclusions.

II. THE SKIP LIST

This section briefly describes the structure and operations of the skip list. The skip list is organized in a multi-layered linked list [31]. Its bottom layer is a standard linked list in which each node points to the next. In the higher layers, nodes point ahead by skipping some of the intermediate nodes. Each node has three components: a pair of key and value, and a set of pointers to succeeding nodes. The nodes are sorted in key order. A skip list of this sort supports a probabilistic binary search over the sorted data, allowing $O(logN)$ insertion and retrieval times. Fig. 1 shows the structure of a basic skip list in

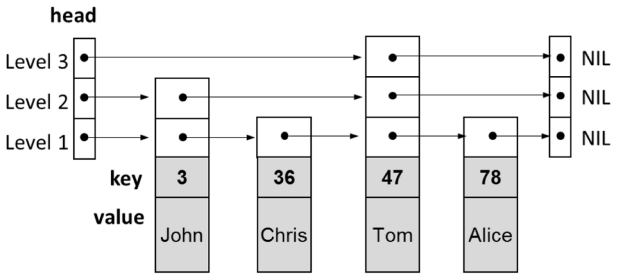

Fig. 1. Initial state of a skip list.

which the maximum height of a node (the number of links) is 3. A skip list supports *PUT*, *GET*, and *DELETE* operations.

A. PUT

The *PUT* operation inserts a new node at an appropriate position in a skip list. Fig. 2 shows how this operation inserts the key '25' with the value 'David'. First, a new node is allocated and initialized with the key and value. The height of the node refers to the number of the links that the node has, and it is determined using a random number generated from a probability function, which makes the likelihood of a node having a particular height 50% of the likelihood that it has one link fewer [32]. This distribution has been shown to produce $O(logN)$ lookup and insertion times [31]. Once the height is decided, the location of the new node in the sorted dataset is found. The node is then added to the list by updating the pointers at all levels. In this example, the key 25 should be placed between key 3 and key 36 and three preceding pointers need to be updated.

A second version of the put operation is the update. In the original formulation of skip lists [31], updates were performed in place. However, the skip list commonly used in commercial software interprets an update as the insertion of a new value for an existing key in front of the previous value, which is never overwritten. This allows all versions of the data to be contained, making it easy to support snapshot isolation [33] through multi-version concurrency control (MVCC) [34], in which all the views are consistent at any point in time. In addition, append-only insertions allow multiple writers to manipulate the data structure concurrently in a thread-safe manner, through atomic operations such as a compare-and-swap (CAS) instruction. Fig. 3 shows what happens when the value of key '47' is updated from 'Tom' by inserting a new node containing the new value, 'Tony', at key '47'.

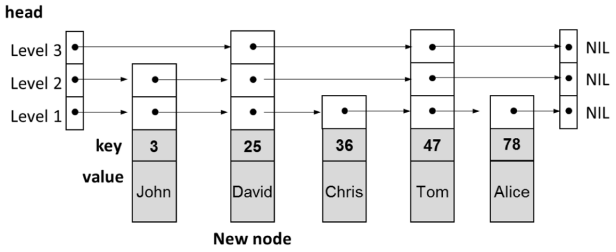

Fig. 2. How a *PUT* operation inserts a new node.

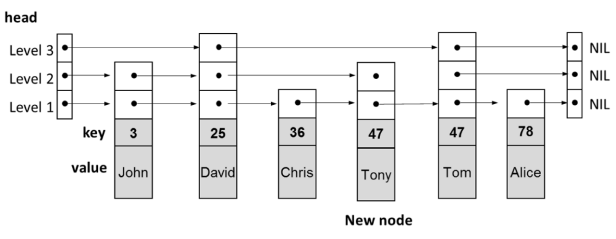

Fig. 3. How a *PUT* operation updates a node by inserting the updated value in a new node.

B. GET

The *GET* operation retrieves the value associated with a given key k by searching the skip list. Starting at the head, the search traverses the list using a stack, comparing k with the key at each node that it encounters. If k is greater than the key at a node, the node is pushed on to the stack and the search follows the link pointing to the next node at the same level. If k is smaller than the key, the search backtracks by popping the previous node from the stack and follows a link at the level below the link that it followed previously. The search concludes when the given key is found and returns the corresponding value.

A search of an append-only skip list cannot terminate reliably when the given key k is found, as it could be only one of the numbers of instances of k, as a result of previous append operations. The search must, therefore, descend to the lowest level of links and identify the first occurrence of k, which will correspond to the most recently inserted value.

Fig. 4 shows a search for the most recent value at key '47'. The search finds a node with a key 47 by traversing the links at the highest level, but it continues to traverse lower levels. At the next level, it proceeds forward until it finds a node with a smaller key than 47, and then moves to the lowest level. When the search finds a node with a key 47, it stops and returns the corresponding value. If the given key does not exist in a skip list, the search stops between the consecutive nodes bracketing the given key k at the lowest level.

C. DELETE

The *DELETE* operation removes a node with a specific key from the skip list. In the original skip list, a deleted node is returned to the free list and the links to and from that node are reconnected. However, in an append-only skip list, a new node is inserted with the same key as the

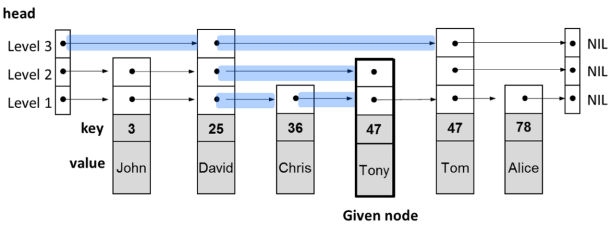

Fig. 4. A *GET* operation on an append-only skip list.

node to be deleted with a special value (often called a tombstone flag). This is intended to provide high concurrency with snapshot isolation among multiple threads and to obviate the need for height rebalancing of the remaining nodes after deletion.

III. A LOW-LATENCY NON-BLOCKING SKIP LIST WITH RETRIAL-FREE SYNCHRONIZATION

We now describe the scalability limitations of existing skip lists and present our low-latency non-blocking skip list.

A. Blocking Skip Lists

When a new data is inserted into a skip list, it is necessary to update a sequence of pointers. If these updates fail to be performed atomically, the structure of the skip list can become inconsistent. Fig. 5 shows an example in which *PUT* operations with keys '25' and '28' arrive at almost the same time. Both the inserted nodes should be placed between the nodes with keys '3' and '36' as the two *PUT* operations are contending for the same group of links. Unless mutual exclusion is enforced on these updates, the skip list may enter an inconsistent state, such as that shown in Fig. 5, in which links (i.e. the colored links) that should point to the same node (i.e. the node with 25) point to the different nodes (i.e. one link points to the node with 28).

To circumvent this undesired outcome, we should be able to update a set of pointers all or nothing, allowing no in-between state visible. A blocking skip ensures the atomicity of multiple pointer updates through the queue-based serialization mechanism; it maintains a wait queue at the front end and places arriving *PUT* requests in the queue. The group of *PUT* operations are popped from the queue and processed serially by a leading thread chosen among the calling threads, while the remaining threads are temporarily blocked. On the completion of the group of *PUT* operations, the leading thread wakes up the associated calling threads so that they can continue.

This single-queue approach addresses an inconsistency problem of a skip list by serializing concurrent updates with a wait queue. However, this benefit comes at the cost of poor scalability; the bottleneck that it introduces severely limits the scalability of a parallel implementation of a skip list.

B. Non-blocking Skip Lists

Many modern processors offer a CAS instruction, which compares the contents of a memory location with an expected value; if they match, it overwrites the contents of the location with a given new value. The availability of this functionality as an operation which is atomic, and therefore cannot be interrupted, allows single pointer updating to be thread-safe without the need for a lock. This is the basis of the non-blocking skip list [9].

The non-blocking skip list ensures the atomic mutation of a series of pointers that need to be updated together based on the CAS operation with some ordering constraints on the updates [9]. Upon a put operation, the non-blocking skip list first searches all the pointers that need to be updated and store them in a queue. Subsequently, a new node can be inserted, and the pointers are retrieved from the queue. By updating the pointers using CAS instructions, it is possible to check that the pointers in the skip list itself still correspond to those in the queue of stored pointers. The pointer updates are enforced from the bottom upwards, and if any pointer has been changed by another thread then the update is aborted and retried, starting with a new search for the remaining pointers. The process is repeated until successful.

Figs. 6 and 7 show an example of two concurrent put operations. Both of them need to create a new node, which is assigned a height of 3, and both of these nodes need to be placed between the keys '3' and '35'. Three preceding pointers need to be updated in both the operations and the consistency of these updates is critical. Fig. 6 shows the timeline of these *PUT* operations in which one thread is interrupted within a critical section. A failure occurs (at timestamp 8) because the second

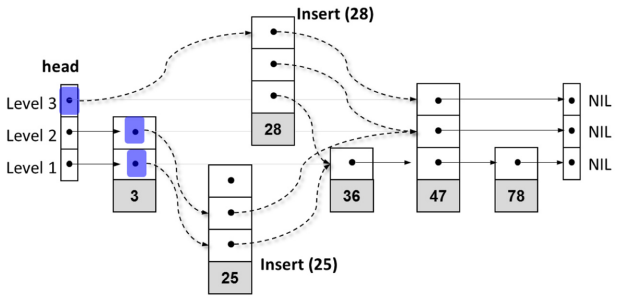

Fig. 5. An inconsistent skip list with non-atomic concurrent writes.

	PUT(25)	PUT(28)
1	Search pointers	
2	CAS (P1)	
3	CAS (P2)	
4		Search pointers
5		CAS (P1)
6		CAS (P2)
7		CAS (P3)
8	CAS (P3) : FAIL	
9	Search pointers	
10	CAS (P3)	

Fig. 6. Timeline of concurrent *PUT* operations in a non-blocking skip list.

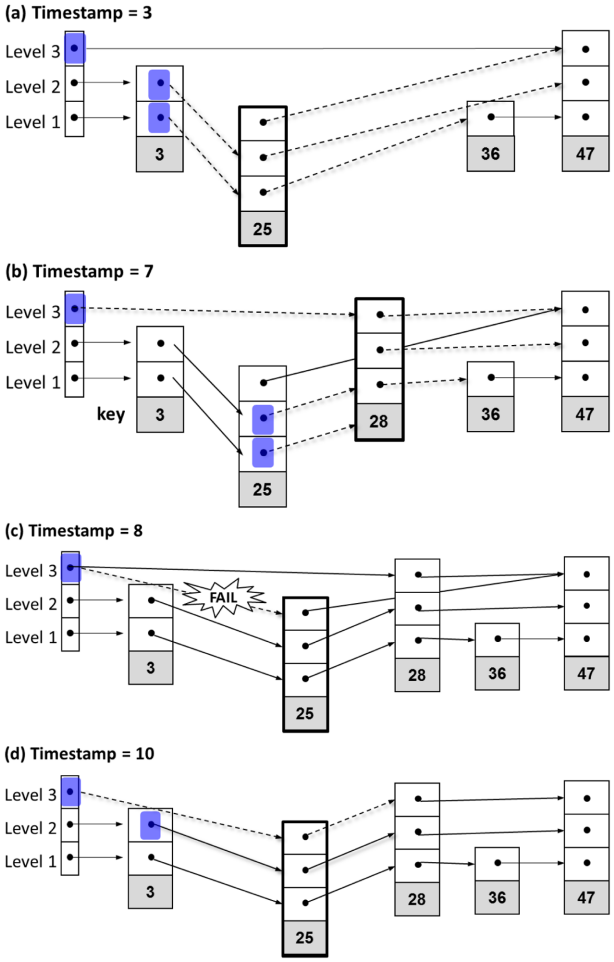

Fig. 7. Snapshots over *PUT* operation timeline.

thread has changed the value of the pointer (P3). The first thread, therefore, resumes operation, with a new search for pointers to be updated (P3).

This stop-and-retry mechanism is effective unless many threads are operating on the same part of a skip list. In that case, threads will continually interfere with each other's attempts to retry link updates, leading to long and highly unpredictable delays. In multi-tenant systems, this long-tail latency can violate service level agreement (SLA).

C. A Low-Latency Non-blocking Skip List

We introduce a new form of skip list, which addresses the issue of long-tail latency, based on the observation that the $O(logN)$ property of a skip list depends on the statistical properties of the heights of all the nodes. The height of some nodes can be reduced to simplify the concurrent operations, provided that compensatory increases in height are made to other nodes, to restore the overall distribution of node heights. Height reduction and

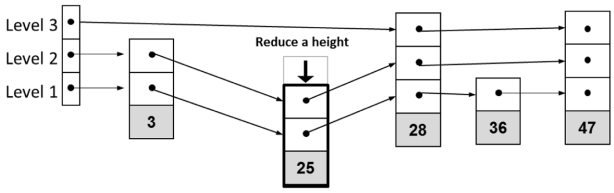

Fig. 8. Consistency support in the LLNB skip list.

statistical compensation are addressed in the following two sections.

1) Post-failure adaption:

The skip list is inherently a probabilistic data structure where the node height (i.e., the number of pointers pointing to the next nodes) is randomly determined by a probability function [32]. The heights eventually have an intended probabilistic distribution in which the $O(logN)$ insertion and lookup times are ensured. The proposed skip list addresses the long-tail latency problem by exploiting this randomness property. We allow *PUT* operations to terminate in failure, despite the fact that all the pointers from a node have not been updated. Instead, we reduce the node height to the number of pointers that have been successfully updated before a failure.

Fig. 8 shows how our low-latency non-blocking skip list copes with a pointer update failure, as demonstrated in the example of Fig. 7. The thread, which has failed to update the highest-level link of a preceding node during a *PUT* operation for key '25' because of interference from the other thread, modifies the height of the node to two, which is the number of pointers that have been successfully updated. This adaptation eliminates the need to retry the updates.

2) Dynamic probability tuning:

One primary challenge of our low-latency non-blocking skip list is how to ensure the intended distribution of node heights against post-failure changes to them in failure; the mutation of the statistical properties of the heights may increase the time complexity of the skip list.

In a skip list, the probability $P(h)$ that a new node n is given a height h is p^h, where p is usually set to 1/2. The height is initialized to 1, and a random number is generated between 0 and 1. If the random number is smaller than p, the height of a new node is set to the current height, which is 1. Otherwise, the height is incremented by 1, and a new random number is generated. Thus, the number of nodes of each height exponentially decreases in proportion to that of height, yielding $O(logN)$ search and insertion times in a skip list.

We need to adapt this procedure for choosing node heights, so as to compensate for the nodes which are reduced in height as a result of failed updates. There is no obvious way of modifying the very straightforward

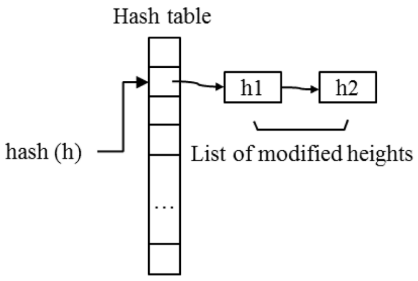

Fig. 9. Hash table containing modified heights.

method of node height determination described above. Instead, we introduce a hash table of node heights that have been replaced and indexed by their node heights (Fig. 9). When a new height is generated by the procedure described above, the hash table is consulted to find out whether any nodes have been modified to that height. If an entry is found, then the new node is given the height of the node that was previously modified. In other words, for each change in height that is used to deal with an update failure, an opposite change is made from the data stored in the hash table.

IV. PERFORMANCE EVALUATION

This section describes the evaluation methodology and experimental results for the performance evaluation of the proposed low-latency non-blocking skip list.

A. Methodology

To assess the effectiveness of our skip list, we implement the original blocking skip list (B-SKIP), the non-blocking skip list (NB-SKIP), and our low-latency non-blocking skip list (LLNB-SKIP). For a comprehensive analysis, we also study the no-hotspot skip list (NH-SKIP), which has been recently designed to improve concurrency through asynchronous link updates [35]. Specifically, the NH-SKIP maintains a background thread that builds indexing links asynchronously, while the foreground writers only insert data into the lowest layer. Accordingly, the NH-SKIP avoids undesired interferences across threads and achieves a fast response time under high concurrency.

Our experiments are performed on a machine with an Intel Xeon 48-core E5-2650 running at 2.2 GHz with a 32 GB main memory. We measure the performance of the skip lists by running a microbenchmark that generates a million *PUT* and a million *GET* operations for three different workloads, each of which is sequential, random, and skewed. The skewed workload mimics a real-world workload that has a locality and is coined based on a Zipfian distribution function provided by a numpy library. The key and value are both set to 16 bytes in size.

B. Throughput

Fig. 10 shows the IOPS (input/output operations per second) of the *PUT* and *GET* operations for the four types of skip lists under various workloads. We measure the performance by varying the number of threads from one to 64.

For *PUT* operations, all the non-blocking skip lists (NB-SKIP, LLNB-SKIP, and NH-SKIP) scale well with the increased number of threads, while the blocking skip list (B-SKIP) exhibits poor performance under high concurrency. The LLNB-SKIP achieves almost identical performance to the NB-SKIP across all workloads. It is apparent that removing retrials for link updates is scarcely effective in improving throughput, and suggests that the distribution of the node heights is effectively maintained by the compensation mechanism in the LLNB-SKIP.

One noticeable result is that the NH-SKIP displays disparate performance trends compared to the other skip lists. It provides excellent performance when the number of threads is smaller than 16, but its performance significantly drops under more threads. Our in-depth analysis reveals that in the NH-SKIP, the indexing links are solely built by a background thread. Thus, if a large number of requests arrive concurrently before the background thread appropriately makes the indexing links, the performance dramatically decreases. In extreme cases, the time complexity of the insertion operation increases to nearly $O(N)$ because the NH-SKIP can linearly search the nodes and does not skip the in-between nodes. Furthermore, the performance of the NH-SKIP is also highly affected by the underlying scheduling policy. If the execution of the background thread is not properly scheduled, the NH-SKIP introduces severe performance fluctuations as shown in the random workloads, which can be a fatal weakness in terms of SLA governance.

Despite the weaknesses above, the NH-SKIP achieves the best performance for all the thread numbers under the skewed workloads. This superiority, however, comes at the cost of sacrificing consistency. As described in Section II, the skip list used in KVS typically supports MVCC, which enables consistent data access without blocking under concurrent reads and writes. To support this property, B-SKIP, NB-SKIP, and LLNB-SKIP do not allow overwrites but insert the updated data as another node. This append-only behavior enables MVCC through snapshot isolation but incurs an overhead cost of inserting and maintaining the duplicated keys. In contrast, NH-SKIP, which is intended for general purposes, has no support for MVCC and simply overwrites an existing

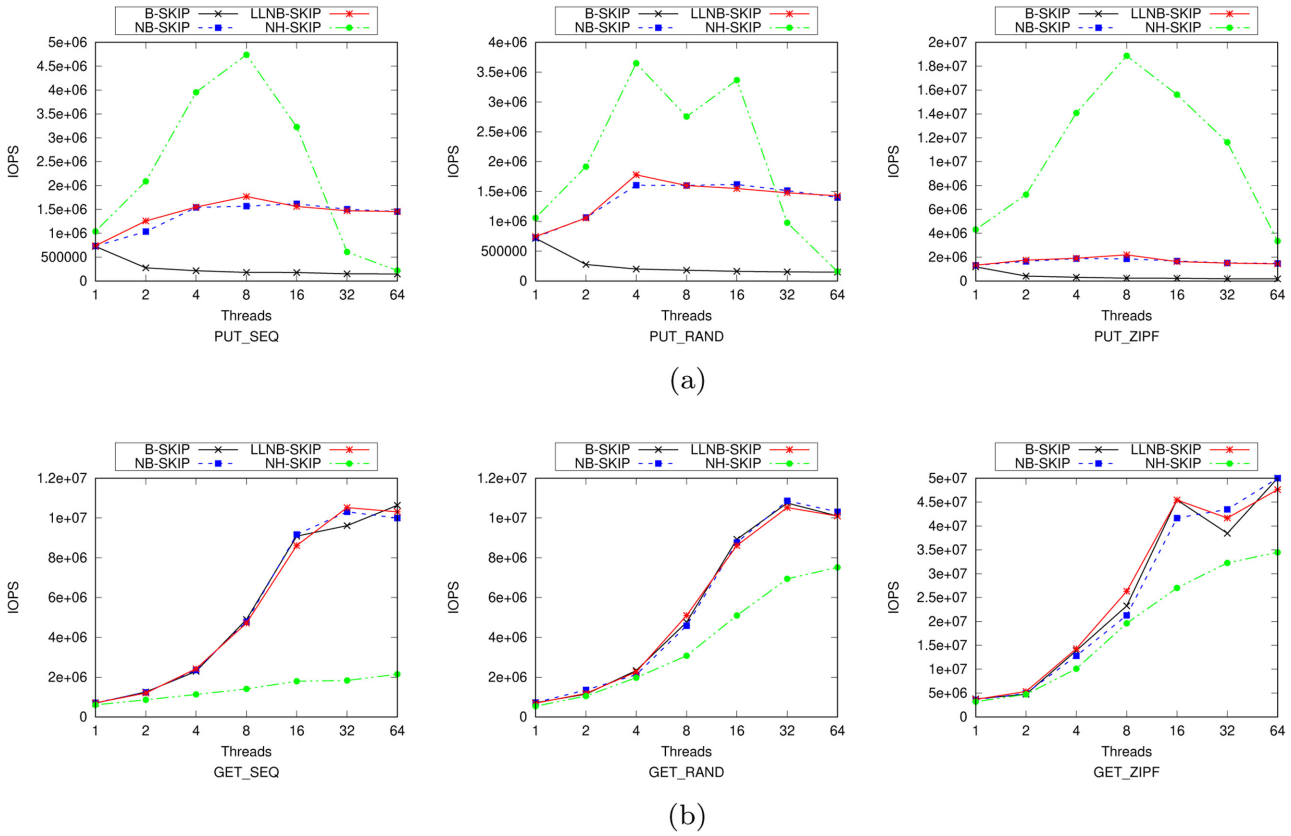

Fig. 10. IOPS of the *PUT* (a) and *GET* (b) operations.

value upon an update, which is less costly than insertion. This relaxation yields a large performance benefit for the skewed workload, as it includes a significant number of updates.

On *GET* workloads, while all of the skip lists are scalable, the degree to which the performance scales varies. There is little variation in performance between the B-SKIP, NB-SKIP, and LLNB-SKIP because they all enable non-blocking reads. In contrast to *PUT* operations, the NH-SKIP provides inferior performance for *GET* operations. The primary reason for this is that the data structures of the NH-SKIP incur more cache misses than others. All the skip lists other than the NH-SKIP use a consecutively allocated table for the node structure, which mimics the skip list design of the commercial-level KVS. However, the NH-SKIP, which is publicly available at [36], uses a linked list to implement a node structure that can dynamically adjust the node height with a background thread. This flexibility adversely affects the performance by increasing the number of cache misses, thereby resulting in an average performance loss of 45%.

C. Tail Latency

To investigate the effectiveness of the proposed techniques in terms of the SLA requirement, we measure the latency of individual requests for different workloads. Fig. 11 shows the cumulative distribution function of the *PUT* latency as the number of threads varies from 4 to 64. The performance at other threads is omitted owing to limited space. In Fig. 11, it can be seen that the LLNB-SKIP reduces tail latency at high concurrency, compared to NB-SKIP. In particular, the LLNB-SKIP is effective in taming tail latency under the skewed workloads, reducing the 99.9th percentile latency by 23% on average, and up to 68% compared to the NB-SKIP. When the workloads have a strong locality, threads contend for the popular group of links, which inevitably increases the latency through more conflicts. Fig. 12 shows the number of CAS failures due to conflicts during concurrent writes and draws similarities to the CDF study. For the sequential workloads, we partition the entire dataset into a disjointed sub-dataset and allocate each sub-dataset to different threads. This reduces the chances of interference of threads with each other and the ratio of conflicts is negligible (less than 0.001%). However, under the skewed workloads where a small set of popular data is frequently accessed, the possibility of occurrence of conflict increases by up to 2%. Consequently, the NB-SKIP incurs a significant number of CAS failures for the

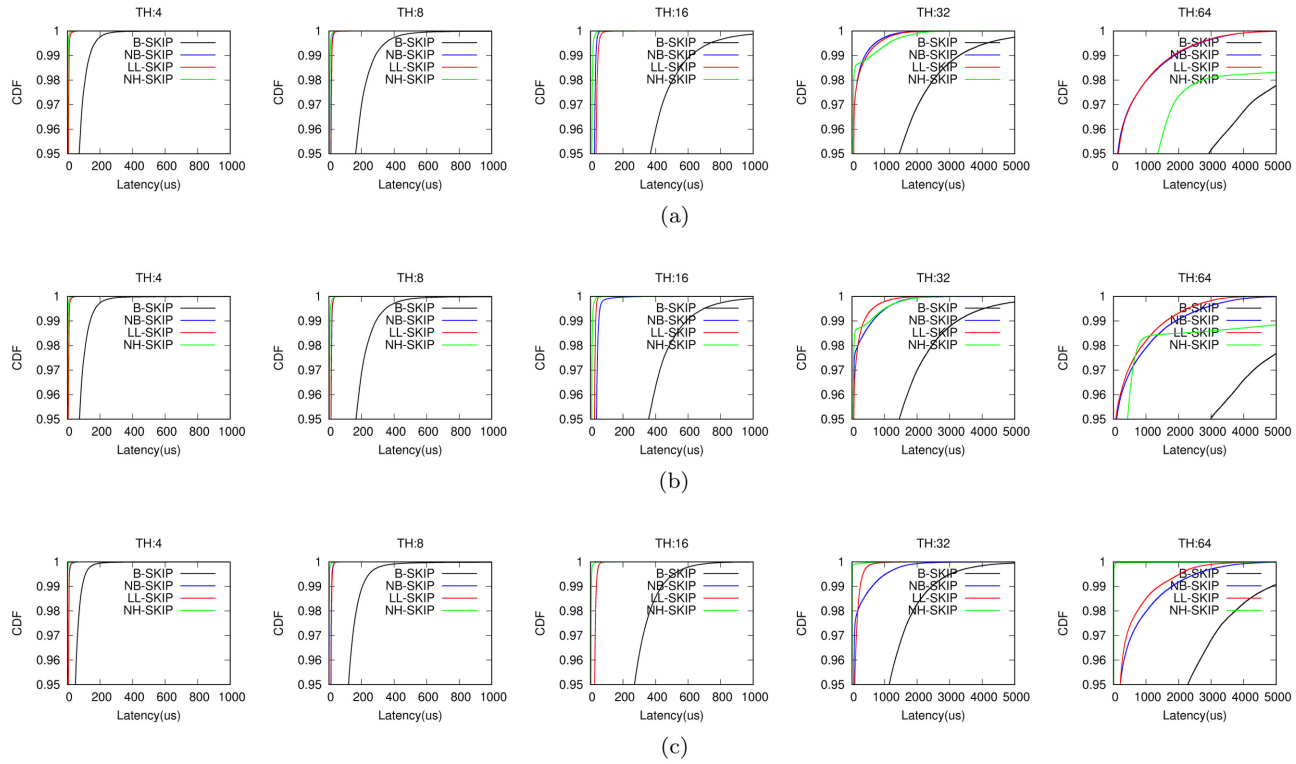

Fig. 11. Cumulative distribution function of the put latency: (a) sequential workloads, (b) random workloads, and (c) skewed workloads.

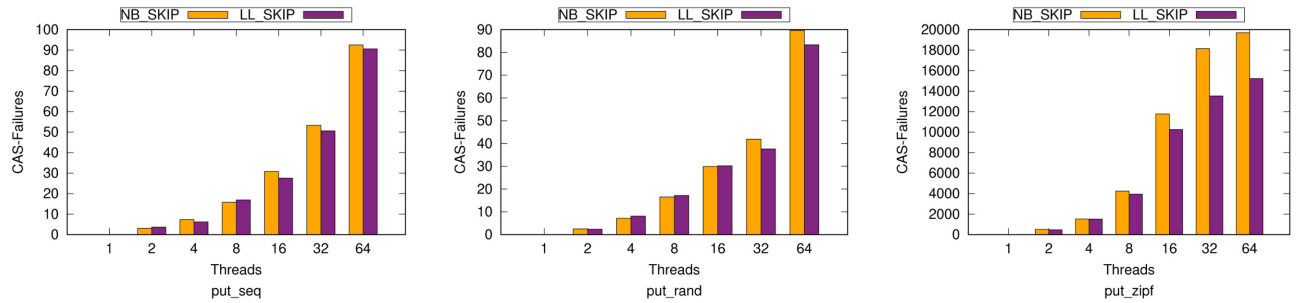

Fig. 12. The number of CAS failures.

skewed workloads that are orders of magnitude larger than those in the other workloads. In contrast, the LLNB-SKIP demonstrates great effectiveness in reducing tail latency in the settings, reducing the number of CAS failures by 13% on average and up to 23% compared to the NB-SKIP.

V. RELATED WORK

There is a long history of modifying data structures for concurrent programming. Michael [5] presents the first implementation of a lock-free hash table by making use of a CAS instruction. Thread-safe insertion and retrieval of data with a hash table are relatively straightforward, but resizing the table is more difficult. This can be approached through data-structure revision and optimization techniques [6, 7]. Michael and Scott [37] invent a queue that supports concurrent enqueue and dequeue operations by separating locks for head and tail accesses using a dummy node. Moir and Shavit [8] present a concurrent linked list that uses a hand-overhand locking mechanism instead of global locking.

Among the previous studies, the most related work includes ones on the scalability of a skip list. The first concurrent lock-free skip list is introduced by Fraser [9] and Harris [10] and its practical implementation by Lea [6] is adopted in the JavaTM SE 6 platform. Herlihy et al. [11] present a lock-based concurrent skip list that ensures the correctness of updates with a momentary locking and

reducing the time delay which a lock needs to be applied. They argue that this approach strikes a balance between complexity and concurrency. Alam et al. [12] introduce a distributed version of a skip list for fast range queries that uses a message-passing mechanism and scales well on a cluster of multi-core computers. Our work shares similarities with these studies, but has notable differences in that we improve the scalability of the skip list in terms of a tail latency, which is a primary concern in multi-tenant systems.

Another class of studies examines the scalability of the Linux systems. A study of MIT investigates the scalability of a large set of applications in multicore systems and reveals a scalability bottleneck in Linux, which they eliminate by introducing a scalable counting technique called a sloppy counter [13]. An operating system structure that mimics a distributed system running over independent cores by means of message passing is proposed [14] to overcome the limited scalability of legacy operating systems. Boyd-Wickizer et al. [15] reveal that the file descriptor access and the memory management codes lack scalability due to their unnecessary sharing across processes. They address this challenge by dedicating cores to specific functions of the operating systems and thus avoiding inter-core bottlenecks.

Scalability has also been actively studied in data management systems. Cheng et al. [16] present a parallel OLAP query processor called 'PhiDB' which efficiently handles concurrent operations with multi-threads on multi-core architectures. Cui et al. [17] also scale OLTP applications to utilize the increasing number of cores in computing systems. They observe that both applications and the Linux kernel have scalability bottlenecks and propose several techniques such as a scalable database lock and a spin lock to address them. Balkesen et al. [18] perform an empirical study on parallel join algorithms (sort-merge and hash join) on multi-core systems. Chhugani et al. [19] modify the merge sort algorithm for multi-core architectures. Han et al. [20] improve the scalability of the KVS by combining two complementary data structures as a unified memory component, and Merritt et al. [21] present a scalable log-structured KVS to support concurrent writes on the multi-core servers with large memories.

VI. CONCLUSION

This paper revealed the scalability limitations of current skip lists extensively used in commercial software. The blocking skip list has poor scalability due to the performance bottleneck caused by queue-based serialization, while the non-blocking skip list suffers from long-tail latency coming from its continual retry behavior to enforce atomicity of a transaction against the interferences of other threads. This paper overcomes this the limitation by presenting a low-latency non-blocking skip list that maintains the consistency of a skip list with retrial-free synchronization. We achieve this goal with two strategies, the post-failure adaptation, which simply reduces the node height and eliminates the need of for a retry in failure, and the dynamic probability tuning, which adjusts the probability to compensate for the resulting reduction in the node heights. The performance evaluation with various workloads demonstrates that our skip list achieves both a scalable performance and low-tail latency under a highly concurrent circumstance. Given that the skip list is widely used in a variety of environments, from Internet service providers such as Google [22] and Facebook [23] to block chain systems such as Hyperledger Fabric [38], it is hypothesized that the proposed low-latency and concurrent skip list will greatly help to improve the user's experience in real systems.

ACKNOWLEDGMENT

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (No. NRF-2017R1D1A1B03031494).

REFERENCES

1. G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp, 1-4, 1965.
2. J. L. Manferdelli, N. K. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808-815, 2008.
3. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Madison, WI: Arpaci-Dusseau Books, 2015.
4. S. H. Fuller and L. I. Millett, *The Future of Computing Performance: Game Over or Next Level?* Washington, DC: National Academy Press, 2011.
5. M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Canada, 2002, pp. 73-82.
6. Doug Lea's Home Page, http://gee.cs.oswego.edu/.
7. O. Shalev and N. Shavit, "Split-ordered lists: lock-free extensible hash tables," *Journal of the ACM*, vol. 53, no. 3, pp. 379-405, 2006.
8. M. Moir and N. Shavit, "Concurrent data structures," in *Handbook of Data Structures and Applications.* Boca Raton, FL: Chapman and Hall/CRC Press, 2004.
9. K. Fraser, "Practical lock-freedom," Computer Laboratory, University of Cambridge, *Technical Report No. UCAM-CL-TR-579*, 2004.
10. T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Distributed Computing*. Heidelberg: Springer, 2001, pp. 300-314.
11. M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A

provably correct scalable concurrent skip list," in *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS)*, Bordeaux, France, 2006.
12. S. Alam, H. Kamal, and A. Wagner, "A scalable distributed skip list for range queries," in *Proceedings of the 23rd International Symposium on High-Performance Parallel And Distributed Computing*, Vancouver, Canada, 2014, pp. 315-318.
13. S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, 2010, pp. 1-16.
14. A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky, MT, 2009, pp. 29-44
15. S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. T. Morris, et al., "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2008, pp. 43-57.
16. X. Cheng, B. He, M. Lu, C. T. Lau, H. P. Huynh, and R. S. M. Goh, "Efficient query processing on many-core architectures: a case study with Intel Xeon phi processor," in *Proceedings of the 2016 International Conference on Management of Data*, San Francisco, CA, 2016, pp. 2081-2084.
17. Y. Cui, Y. Chen, and Y. Shi, "Scaling OLTP applications on commodity multi-core platforms," in *Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, White Plains, NY, 2010, pp. 134-143.
18. C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu, "Multi-core, main-memory joins: sort vs. hash revisited," *Proceedings of the VLDB Endowment*, vol. 7, no. 1, pp. 85-96, 2013.
19. J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313-1324, 2008.
20. Y. Han, B. S. Kim, J. Yeon, S. Lee, and E. Lee, "TeksDB: weaving data structures for a high-performance key-value store," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, article no. 8, 2019.
21. A. Merritt, A. Gavrilovska, Y. Chen, and D. Milojicic, "Concurrent log-structured memory for many-core key-value stores," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, pp. 458-471, 2017.
22. Leveldb, https://github.com/google/leveldb.
23. rocksdb, https://github.com/facebook/rocksdb.
24. Apache, "HBase," https://hbase.apache.org.
25. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, No. 6, pp. 205-220, 2007.
26. What is Memcached?, https://memcached.org.
27. Redis, https://redis.io.
28. The skiplist data structure, https://lwn.net/Articles/552088.
29. B. Liu, "Btrfs: apply the probabilistic skiplist on btrfs," 2012, https://marc.info/?l=linux-btrfs&m=132618090507787&w=2.
30. A kernel skiplist implementation (Part 1), https://lwn.net/Articles/551896/.
31. W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668-676, 1990.
32. T. Papadakis, "Skip lists and probabilistic analysis of algorithms," Ph.D. dissertation, University of Waterloo, Canada, 1993.
33. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1-10, 1995.
34. P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 185-221, 1981.
35. T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *Proceedings of 2013 IEEE 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, 2013, pp. 196-205.
36. V. Gramoli, "Synchrobench," https://github.com/gramoli/synchrobench.git.
37. M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1-26, 1998.
38. IBM, "Hyperledger Fabric state database," https://hyperledger-fabric.readthedocs.io/en/release-1.4/ledger.html.

Eunji Lee https://orcid.org/0000-0001-5916-2301

Eunji Lee received a Ph.D. degree in computer engineering from Seoul National University in 2012. She was a visiting scholar in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, MI, and a senior engineer at the Memory Division of Samsung Electronics, Co. Ltd. She is currently an assistant professor in the Department of Smart Systems Software, Soongsil University, Seoul, South Korea. Her research interests include cloud computing, data analytics platforms, data-centric applications, and emerging memory technologies. She has published more than 40 papers in leading conferences and journals in these fields, including *the IEEE Transactions on Computers*, *the IEEE Transactions on Knowledge and Data Engineering*, and *ACM Transactions on Storage*. She also received the Best Paper Awards at the USENIX Conference on File and Storage Technologies in 2013.